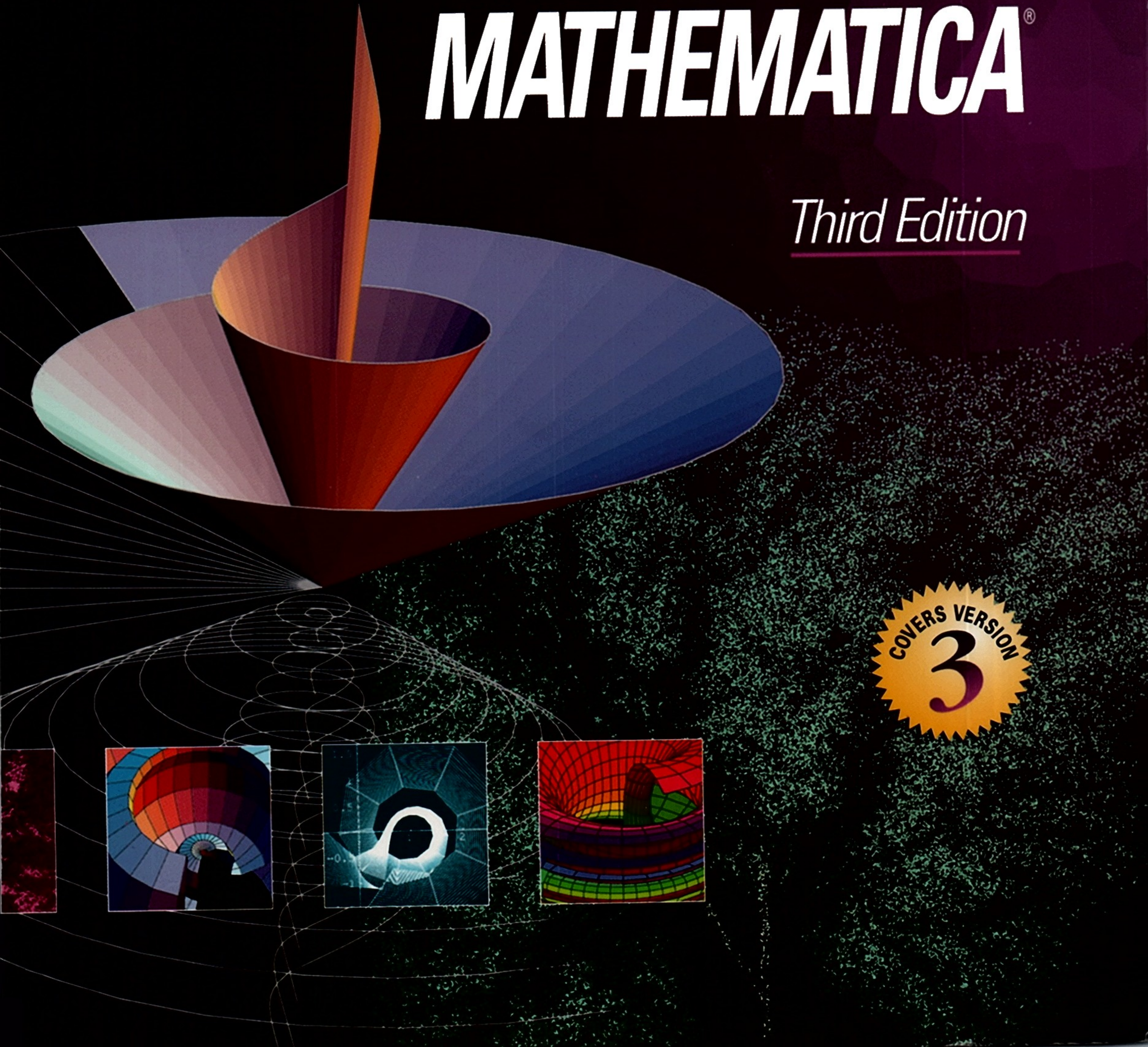
ROMAN MAEDER

PROGRAMMING IN MATHEMATICA®

Third Edition



Programming in Mathematica®

Third Edition

Programming in Mathematica®

Third Edition

Roman E. Maeder

ADDISON-WESLEY
An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

The publisher offers discounts on this book when ordered in quantity for special sales.

For more information, please contact:

Corporate & Professional Publishing Group
Addison-Wesley Publishing Company
One Jacob Way
Reading, Massachusetts 01867

Library of Congress Cataloging-in-Publication Data

Maeder, Roman.
Programming in Mathematica / Roman E. Maeder. — 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-85449-X
1. Science—Data processing. 2. Mathematica (Computer programming language). I. Title.
Q183.9.M34 1996
510'.285'53--dc20 96-5714
CIP

This book was prepared by the author, using the TEX typesetting language on a Sun SPARCstation 5 computer.

ISBN 0-201-85449-X

2 3 4 5 6 7 CRW 01 00 99 98

2nd Printing July, 1998

■ Preface

Mathematica was officially announced in June of 1988. Since then it has found many uses in very diverse fields. While it is useful to do a few calculations interactively, its real strength lies in the programming language it offers. Writing programs, one extends *Mathematica* with specialized new functions in one's own field of interest. *Mathematica*'s programming language is unlike any you have encountered before. The language's manual explains all its features and gives some very basic examples of their use. For writing good programs, however, this is not enough: there is clearly a need for a book explaining all the many features in context and giving more extensive examples of their use. The first edition of this book filled this gap for version 1.2 of *Mathematica*. Like the second edition, this third edition was prompted by a major new version of *Mathematica*, Version 3.

I have been in contact with many early users of *Mathematica* and I have also given students at the University of Illinois and the Swiss Federal Institute of Technology an opportunity to learn about it. I have seen many programs written in *Mathematica*, good ones and bad ones. The bad ones invariably solve a particular problem in an unnecessarily complicated way, perhaps because the programmers were unaware of the elegant constructs available in *Mathematica*. Let me point out these constructs and show you how they can be used to write concise, elegant, and efficient programs.

The best way to teach these *Mathematica*-specific programming methods is to look at examples of complete programs that solve some nontrivial problem. Even if the example chosen does not lie in the particular field of application that you are interested in, you will be able to use similar ideas for your own programs. Many of the examples presented here deal with graphics. Graphics applications are especially suited for learning programming, because it is easy to see whether your code is correct simply by looking at the picture it produces. Advanced graphics applications are also sufficiently challenging to require advanced programming methods. Other examples come from symbolic computation, geometry, dynamics, and numerical mathematics. Along the way, we also develop many short pieces of code that can be used as parts of larger programs or that help to customize *Mathematica* to your particular needs. Many of these found their way into the standard *Mathematica* distribution and are now available to all users.

Because there exists an excellent manual, this book does not explain everything from scratch, but assumes some familiarity with *Mathematica*. I assume also that you have access to a computer on which to try out the examples. In this book I want to tell you how you can use *Mathematica* for more than just typing in single commands. If you are using *Mathematica* for your work, teaching, or solving homework assignments, sooner or later you will encounter more advanced problems requiring many commands to solve, or you will be faced with doing the same calculation steps over and over again with different input. This is the point at which you want to start writing *programs*. *Mathematica* includes a rich and powerful programming language. Unlike the usual languages such as BASIC and **C**, it is not restricted to a small number of data types, but allows you to perform all its symbolic

computations. The *Mathematica* manual can only hint at the possibilities. It explains all the features but does not show you which ones to use for a particular problem or how to fit things together into larger programs.

Through design and tradition, each programming language has developed a certain preferred style of good programming. It is possible to solve the same problem in many different ways, but there is usually some idea about what is a good or a bad program. In this book, I want to present examples of what the designers of *Mathematica* think is good programming style and show you why this is so. Even if you write your programs strictly for personal use, you will benefit from following a good style. For developing programs for others to use, adhering to this style is indispensable.

The programming examples in this book serve two purposes. First, they help explain concepts and show how things fit together to make up complete programs. Second, they are designed to be more than mere toy programs and should prove useful in their own right. In developing an example we always use the same method. We start out with a few commands or definitions that could be entered directly into *Mathematica*. We then extract the parts of the computation that are the same regardless of the input and define some functions or procedures that automate these steps. Then we apply standard techniques to these functions to make them into a *package*, adding documentation and certain programming constructs that make such a package easier to use. The goal is to write a program that would be useful not only to its author, who knows how it works, but also to other people. Finally, we might add a few more functions to the package or look at alternatives to what we did so far. In Chapter 1 these steps are described in full detail. Later on, when we concentrate on other aspects, we assume that you are familiar with these basic concepts and we shall not mention all the steps in detail. All the programs developed in this book are either part of the standard *Mathematica* distribution from Wolfram Research, or they are available free in electronic form.

Version 3 brings major new features. The programming language itself has not changed all that much, but many problems that required elaborate workarounds and many inconsistencies have been remedied. As a consequence, it is now possible to present the material on packages in a more logical fashion. The support for developing larger applications has been improved, and this edition discusses the software engineering issues of writing and using larger programs in *Mathematica*. The treatment of exact numeric quantities is another area of improvement, and the way numerical code should be written has changed. All programs have been revised to take advantage of the many new built-in functions. I added more material about functional and structural programming. These techniques are fundamental to writing good *Mathematica* code, but they are not available in most other programming languages and therefore need an expanded treatment.

The most important addition to Version 3 is, of course, the new frontend and the typesetting capabilities. The fact that notebooks and typeset formulae are represented as ordinary *Mathematica* expressions means that they can be manipulated easily with programs. This capability leads to some fascinating applications, and the new material on the frontend and typesetting teaches you how to develop such applications.

A complete, larger application (iterated function systems), more exercises, and an

updated bibliography complete this expanded and revised edition.

This book is no replacement for the *Mathematica* manual "The *Mathematica* Book" [40]. I do not expect that you have read everything in the *Mathematica* book, but you should have some basic experience with *Mathematica* before reading this book. Single commands are usually used without detailed explanation. You can use the index in the *Mathematica* book to look up a description of a command that you did not know about. We also give references to places in the *Mathematica* book where you can find explanations of concepts that are particularly relevant to a topic in this book. You should always turn to the *Mathematica* book for explanations of features that are assumed known here, but that you have not used yet. The place to look for an explanation of all variants, defaults, or options for a particular command is the *reference guide* in the back of the *Mathematica* book. All this information is also available in electronic form and can be accessed through the Help Browser of *Mathematica*.

All explanations about how *Mathematica* works are based on Version 3. The first edition of this book was about Version 1.2. Many things have changed in the new version. I added a few sections called "Changes from Earlier Editions" for the benefit of readers familiar with earlier versions of *Mathematica* or the first and second edition of this book.

I am grateful to many people who have contributed to this book. My thanks go first to the other developers of *Mathematica*. The language was shaped through countless discussions and many heated arguments. Trying to explain to each other *why* we think a certain feature should be done in a certain way has deepened our understanding of the matters involved and has given the language its overall consistency, despite the fact that it contains hundreds of commands and unifies many diverse programming paradigms.

Helpful ideas for this book came from Jim Feagin, Theodore Gray, Dan Grayson, Jerry Keiper, Silvio Levy, Troels Petersen, Will Self, Bruce Smith, Ilan Vardi, Ferrell Wheeler, and Stephen Wolfram. Some of the examples I used were inspired by Henry Cejtin, John Gray, Lee Rubel, William Thurston, Ilan Vardi, Jörg Waldvogel, and Dave Withoff. Help with the typographical side of producing this book came from Peter Altenberg, John Bonadies, Joe Kaiping, Daniel Lee, Cameron Smith, and Gregg Snyder, as well as from many other people at Wolfram Research, Inc. and the staff of Addison-Wesley. My past and present publishers, Allan Wylde and Peter Gordon, encouraged me to get started on this book and this new edition.

R. E. M.

Herrliberg, Switzerland
August 1996

A computer, to print out a fact,
Will divide, multiply, and subtract.
But this output can be
No more than debris,
If the input was short of exact.
– Gigo

■ Contents

■ About This Book

This section gives you hints and recommendations for benefitting the most from the material in this book.

■ Chapter Overview

Chapter 1 develops a package from scratch. It starts with a few simple interactive commands and then explores ways to combine such commands into small programs. Along the way we learn how to set up package contexts, define defaults for parameters of functions, and also look at graphics. The example chosen comes from mathematics: functions of one complex variable. You can follow the material even if you are not familiar with the mathematical concepts.

In Chapter 2 we look at the theory of writing good packages. The concepts introduced here are used throughout the rest of the book. Understanding why things should be done in a certain way might be interesting on its own, but you can also learn to apply the concepts in a cookbook-style way. A skeletal package is provided, which you can use as a template for writing your own packages.

Chapter 3 looks at issues of pattern matching and defining options for your functions. Default values for parameters and options can make commands much easier to use. *Mathematica* itself makes heavy use of these features.

Chapter 4 looks at different programming styles possible in *Mathematica*. It is here that the preferred functional style is explained. Understanding this chapter allows you to write better programs.

Chapter 5 looks at some aspects of how expressions are evaluated. If you want to write sophisticated rules and definition,s you need to know about these things. Otherwise, you can skip this material and return to it later as needed.

In Chapter 6 we introduce *mathematical programming*, a way of expressing mathematical relations and formulae that is unique to *Mathematica*. If your application requires simplification and transformation of symbolic expressions, you should read this chapter.

Numerical computations are the topic of Chapter 7. It explains how numerical computations are done and what kind of numbers *Mathematica* supports. You should consult it as needed.

Mathematica has built-in rules and procedures for performing certain transformations automatically. Advanced applications may make it necessary to change these built-in rules or add others. Chapter 8 explains how this can be done. You should have read Chapter 5 before studying the material in here.

Chapter 9 treats input and output. *Mathematica* interacts with the rest of your computer system in many ways, and this chapter deals with this interface. A discussion of typesetting mathematical expressions in the frontend is also contained in this chapter.

Chapter 10 is about graphics programming. It presents some graphic utilities and looks at the package ParametricPlot3D.m and at animations of graphics.

Chapter 11 looks at *notebooks*, available with the frontend for *Mathematica*. It looks at the relationship between notebooks and packages. If you use notebooks or if you want to write programs that run on machines with or without the notebook interface, you should read this chapter. The frontend is programmable; we show some useful applications of frontend programming.

Chapter 12 presents a larger application, iterated function systems. These systems of affine maps are one aspect of the study of chaos and dynamical systems. Their implementation in *Mathematica* poses a number of interesting programming problems.

Appendix A contains programming exercises and their solutions. Appendix B is an annotated bibliography for further reading about the topics of this book, and it contains the list of references.

Some sections are marked as "Applications." They introduce a programming example that makes use of the concepts covered in the preceding sections. They are independent of the rest of the text. Some sections are marked "Advanced Topic." They have a higher level of difficulty than the rest of the book or require higher mathematics. They, too, are optional. The sections "Changes from Earlier Editions" are intended for readers familiar with earlier editions of this book. They explain some of the major changes introduced with Version 3 of *Mathematica*.

■ About the Examples

All examples were tested with Version 3 of *Mathematica*. The "live" calculation sequences in this book were computed on a Sun SPARCstation 5 running SunOS 4.1.4. The examples will not work with earlier versions of *Mathematica*. Those examples which interact with the operating system of the computer on which *Mathematica* is running are, of course, machine dependent and will look completely different on computers that do not run UNIX. Listing init.m shows the commands that have been put into the initialization file init.m for computing the examples in this book.

Some of the packages in this book depend on other packages. To read them into *Mathematica*, you must have all of these imported packages available, too.

In the example *Mathematica* sessions in the later chapters, we generally no longer show the command to read in the package that is the topic of the example. This command of the form `<<ProgrammingInMathematica`` *Package* ` is assumed at the beginning of every session that uses functions from the package in question. Even better is `Needs["ProgrammingInMathematica`` *Package* `"]`, which avoids loading a package more than once.

Many examples developed in this book are now part of the standard *Mathematica* packages. They can be found in subdirectories of AddOns/StandardPackages inside your *Mathematica* installation. All remaining example programs and notebooks are part of the *Mathematica*, Version 3, distribution from Wolfram Research. You can access them directly

```
SetOptions["stdout", PageWidth->58] (* line width *)

Format[Continuation[_]] := ""       (* no blank lines *)

SeedRandom[10000]                   (* reproducible "random" numbers *)

Off[ General::spell, General::spell1 ]

SetOptions[ ParametricPlot3D, Axes -> None ]

Needs["ProgrammingInMathematica`Options`"]
SetAllOptions[ ColorOutput -> GrayLevel ]

$DefaultFont = {"Times-Roman", 9.0}

Begin["`Private`"]
Unprotect[Short]
Short[e_] := Short[e, 2]            (* lines are very short *)
Protect[Short]
End[]

Null
```

init.m: *Mathematica* initialization for this book

from the *Mathematica* CD-ROM or install them onto your hard disk using the *Mathematica* installer. They will be installed into the directory AddOns/ExtraPackages/ProgrammingInMathematica. A list of all programs is shown on page 355.

If the ProgrammingInMathematica directory is installed in AddOns/ExtraPackages the help browser of *Mathematica* may be unable to find the on-line help provided. In this case you should move the directory to AddOns/Applications or put the directory AddOns/ExtraPackages on the help path using the frontend's option inspector (item Global Options ▷ File Locations ▷ AddOnHelpPath).

■ Notation and Terminology

A package is identified by a name (the context name, as we shall see), for example `ComplexMap`. The files in which we store successive versions of this package will be called ComplexMap1.m, ComplexMap2.m, and so on. The final version will then simply be called ComplexMap.m. Much of the text in successive versions of the same package is the same, and we do not generally reproduce it in full.

Mathematica input and output is typeset in typewriter-like style: `Expand[(x+y)^9]`. Parts of such input that are not literal, but denote (meta-)variables, are typeset in italics: `f[`*var*`_] :=` *body*. *Functions* or *commands* are referred to by their name followed by an empty argument list, for example `Expand[]`. Listings of programs are delimited with horizontal lines and usually have captions beneath them. Major listings, tables, and figures are numbered by section. The numbers have the form $c.s$–n, where c is the chapter number, s is the section number, and n is a consecutive number within a section.

Names of files containing programs are typeset in this boldface style: ParametricPlot3D.m. Genuine dialogue with *Mathematica* is set in two columns. The left column

contains explanations and the right column contains the input and output, including graphics. You should be familiar with these conventions from the *Mathematica* book.

In the programming examples, I have tried to follow a uniform style for the indentation of lines in a definition. Because *Mathematica* allows you to write deeply nested expressions, lines are often rather long and have to be broken up so as to fit on the printed page.

In most programming languages you can define *procedures*, *subroutines*, or *functions*. In *Mathematica*, all of these are just another way of looking at *definitions*, commands of the form `f[x_] := body`. Often we also use the term *global rule*. These terms are used interchangeably, depending on which point of view we want to emphasize in a particular situation. Rules proper are expressions of the form `lhs -> rhs` or `lhs :> rhs`. A *substitution* is an expression of the form `expr /. rule`.

■ The *Programming in Mathematica Web Site*

The World Wide Web (WWW) site `<http://www.wolfram.com/Maeder/ProgInMath>` is a repository of information relating to this book. Among other things, you will find errata, program updates, and links to further information of interest to readers of this book, as well as an email address and feedback form for contacting the author. You can access this information using any WWW browser, such as Netscape, Microsoft Internet Explorer, or Mosaic.

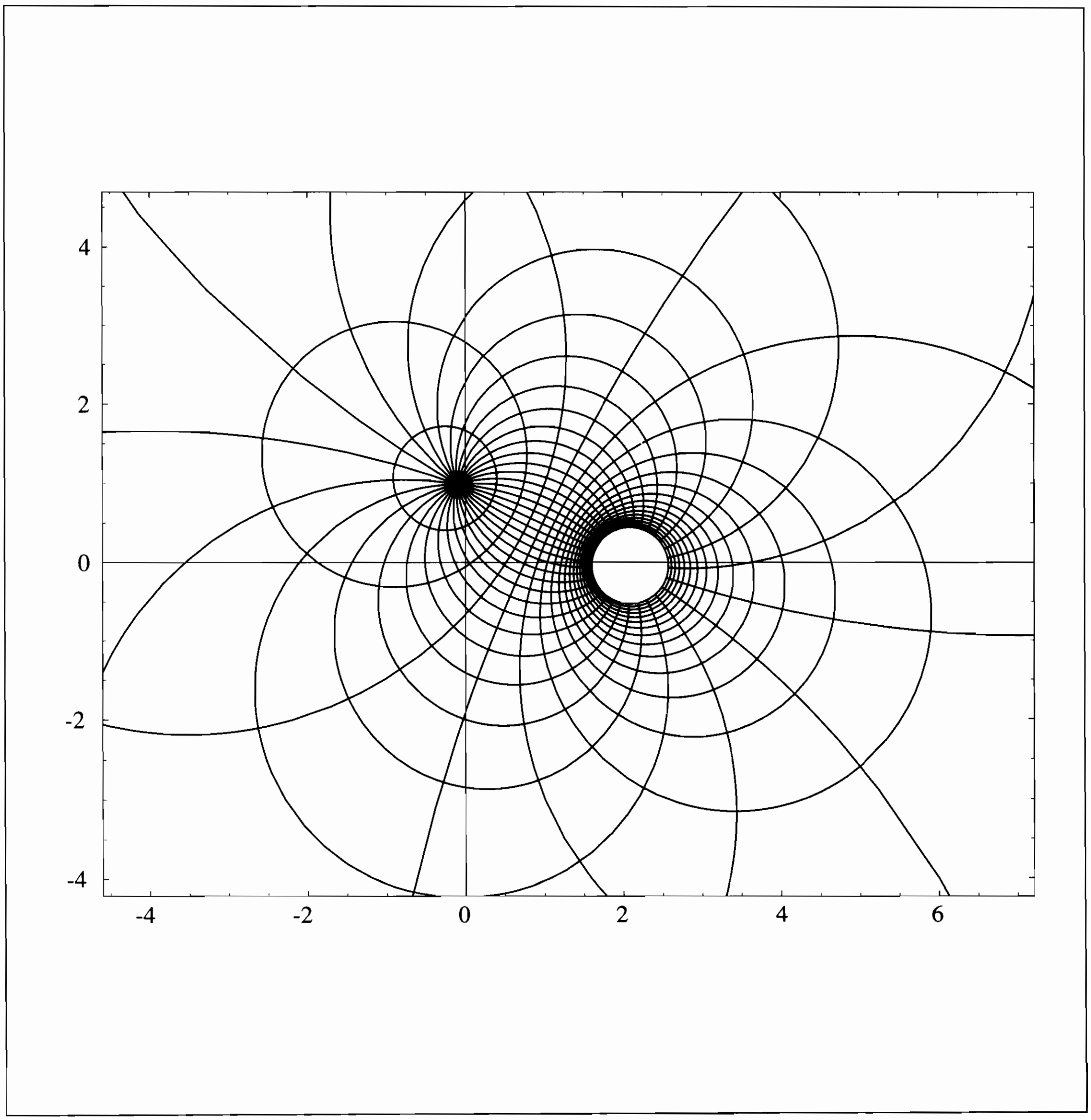
■ Teaching *Mathematica* Programming

Although not designed primarily as a textbook for classes about programming or the use of *Mathematica*, this book can be used in advanced classes that want to give a thorough introduction to *Mathematica*. Such a course should use computers and encourage students to *experiment*. Programming projects selected from the background of the students (from any science) are more valuable than artificial small (toy) programs, because they emphasize the use of *Mathematica* as a tool for doing research in any area rather than as something to learn for its own sake.

Chapter 1 of the *Mathematica* book should be treated first. The step-by-step example in our Chapter 1 provides the motivation for learning good programming style. If time permits, Chapter 2 should be treated in some detail, at least Section 2.1. The template package Skeleton.m from Section 2.4 or the notebook Template.nb from Section 11.1 can be used as a starting point for all programs the students develop themselves. Functional programming is fundamental to good programming style and the first four sections of Chapter 4 should be treated in detail. Depending on the applications in mind, Chapters 6, 7, or 8 should be looked at next. The rest of the book can be consulted as a reference when needed. At this point in a course the emphasis should be put on developing larger applications. Such courses have been developed even before *Mathematica* was available, using a variety of other programs. They can easily be adapted to *Mathematica*. The bibliography in Section B.1.3 gives some references.

Chapter 1

Introduction



In this chapter we shall develop a sample package from scratch. We start out with commands entered interactively into *Mathematica*. Then we collect them into a definition in which the parts we would like to modify are defined as parameters to a replacement rule. This rule will look like a traditional procedure definition in any of the common higher level programming languages.

If you save this definition in a file to read it into *Mathematica* by a single command, you have written your first "package." We shall then add commands to set up a separate context for the functions defined in the package. This will isolate any local variables and auxiliary functions used in the implementation of the package. It will make only those objects visible that are to be exported from the package. These are the functions and variables that you can use after having read the package into your current *Mathematica* session.

Next, a second function is added to the package. Some of the code common to both functions will be put into a separate auxiliary function. This saves space and makes the program easier to maintain. If a change is required, it has to be done in only one place and will be consistently used everywhere.

Then it will be time to add some useful extra features to the basic algorithms. We shall define default values for frequently used parameters. Another area of concern is the graceful handling of bad user input to one of our functions. We shall deal with arithmetic exceptions such as division by zero.

The example chosen for this chapter comes from mathematics. We shall draw graphs of complex-valued functions. However, you do not need to know much about the mathematical properties of complex numbers to understand this example. The emphasis is on the programming part.

Sections 1 through 3 will get you from the interactive session to a fully developed package that is useful in its own right. The following sections introduce refinements that should be done for any package you want to use often or make available to others.

About the illustration overleaf:

A picture of a Möbius transform generated with the command

```
PolarMap[ (2# - I)/(# - 1 + 0.1I)&, {0.001, 5}, {0, 2Pi},
       Frame -> True, Lines -> {20, 36}, PlotPoints -> 40 ]
```

This command (it is not built-in) is developed in this chapter.

1.1 From Calculations to Programs

This chapter shows the stages in building up a package in *Mathematica*. To show how things really work, we shall use a real example, complete with all the necessary details. The package we choose is one for plotting functions of complex variables. Built-in is the `Plot[]` function (see Section 1.9 of the *Mathematica* book), which can plot functions of one real variable.

Most of the functions (logarithms, exponential functions, trigonometric and inverse trigonometric functions) encountered in high school and college mathematics can be evaluated for complex-valued arguments as well. Because there is no built-in command for plotting functions with complex values, we have to add one ourselves.

A complex number consists of two components: the *real* and *imaginary* parts. They can be viewed as the coordinates of a point in the plane. The complex number describing the point with coordinates (a, b) is written as $a + ib$, where i is the *imaginary unit* and stands for $\sqrt{-1}$. In *Mathematica*, we have to write it as `I` instead of `i`, because all built-in symbols begin with a capital letter.

Because plotting a function of a complex variable would need four dimensions (two for the complex variable and two for the complex function value), we need a different way of visualizing such a function. One way of drawing a complex-valued function is to show how the coordinate lines are transformed under it. In Cartesian coordinates, a complex number is written as $x + iy$. The coordinate lines are the horizontal lines of constant y and the vertical lines of constant x. Each of these lines is mapped to a curve in the complex plane under a function f. If we let y be fixed, $f(x + iy)$ is the formula for a curve in the complex plane with parameter x, and vice versa for fixed x and parameter y. If f is the identity (that is, $f(x + iy) = x + iy$), the lines are not deformed and we should see a picture of the coordinate lines themselves. This picture is the first one that we want to draw.

1.1.1 A Plot of the Coordinate Lines

To draw coordinate lines, we determine the coordinates of the points $x + iy$, and then draw two sets of lines. In the first set, we successively set y to certain values; the results are formulae of horizontal straight lines (because x is varied). For the second set, we set x to certain values to get formulae for vertical lines. Even though certain of these steps (such as determining the real and imaginary part of $x + iy$) are trivial, we perform them in full generality, because we want to use the same method later on for arbitrary functions $f(x+iy)$.

Here is the expression for which we want to draw the lines.

```
In[1]:= z = x + I y
Out[1]= x + I y
```

The coordinates of the complex number z are the latter's real and imaginary parts.

```
In[2]:= cz = {Re[z], Im[z]}
Out[2]= {-Im[y] + Re[x], Im[x] + Re[y]}
```

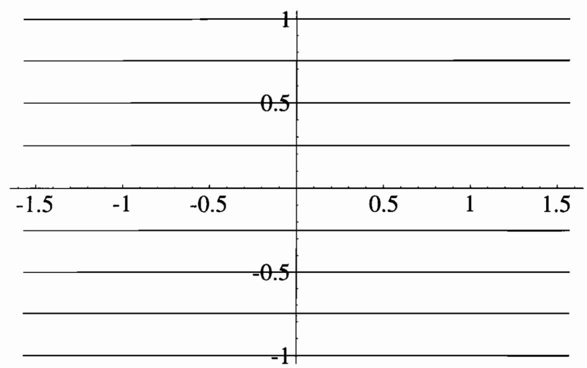
This command generates a list of formulae for nine horizontal lines in the complex plane.

```
In[3]:= hlines = Table[ N[cz], {y, -1, 1, 2/8} ]
Out[3]= {{Re[x], -1. + Im[x]}, {Re[x], -0.75 + Im[x]},
  {Re[x], -0.5 + Im[x]}, {Re[x], -0.25 + Im[x]},
  {Re[x], Im[x]}, {Re[x], 0.25 + Im[x]},
  {Re[x], 0.5 + Im[x]}, {Re[x], 0.75 + Im[x]},
  {Re[x], 1. + Im[x]}}
```

This command plots the lines. `ParametricPlot[]` evaluates the formulae for several values of x in the range $-\pi/2 \leq x \leq \pi/2$, and connects the points by straight line segments.

Because `ParametricPlot` evaluates its arguments in a nonstandard way, we need to force evaluation of the variable `hlines` by wrapping it into `Evaluate[]`.

```
In[4]:= ParametricPlot[ Evaluate[hlines],
                        {x, -Pi/2, Pi/2} ]
```




```
Out[4]= -Graphics-
```

The lines generated constitute the first element of the resulting data structure `-Graphics-`.

```
In[5]:= hg = %[[1]];
```

This command generates formulae of 13 vertical lines. We suppress the lengthy output by placing a semicolon at the end.

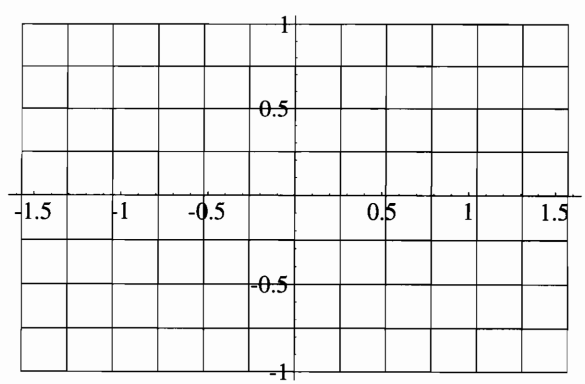
```
In[6]:= vlines = Table[N[cz], {x, -Pi/2, Pi/2, Pi/12}];
```

This time, we want to generate the graphics without displaying it. We are interested only in the lines themselves.

```
In[7]:= vg = ParametricPlot[ Evaluate[vlines],
              {y, -1, 1}, DisplayFunction->Identity
           ][[1]];
```

Now, we combine the two sets of lines to create one picture of the Cartesian coordinate lines. The terminating semicolon suppresses the output of `-Graphics-`, which does not interest us.

```
In[8]:= Show[ Graphics[Join[hg, vg]],
              AspectRatio->Automatic, Axes->True ];
```



The option setting DisplayFunction->Identity causes the graphics functions Plot[], Plot3D[], ParametricPlot[], and so on to generate the graphics in the normal way, but not to render the images. We use it if we want to manipulate the resulting graphics further. Afterward, we can render the images with Show[*graphics*, DisplayFunction->$DisplayFunction].

1.1.2 A Picture of the Sine Function

If we replace the numbers $x + iy$ by $\sin(x + iy)$, we immediately get a picture of the coordinate lines under the sine function.

Here is the expression for which we want to draw lines.

```
In[1]:= z = Sin[x + I y]
Out[1]= Sin[x + I y]
```

The coordinates of the complex number z are again obtained as the real and imaginary parts.

```
In[2]:= cz = {Re[z], Im[z]}
Out[2]= {Re[Sin[x + I y]], Im[Sin[x + I y]]}
```

This command generates formulae for the images of the horizontal lines under the sine function.

```
In[3]:= hlines = Table[N[cz], {y, -1, 1, 2/10}];
```

Again, we generate the graphics without rendering the images.

```
In[4]:= hg = ParametricPlot[ Evaluate[hlines],
                {x, -Pi/2, Pi/2},
                DisplayFunction->Identity
              ][[1]];
```

Here are the formulae for the image of the vertical lines.

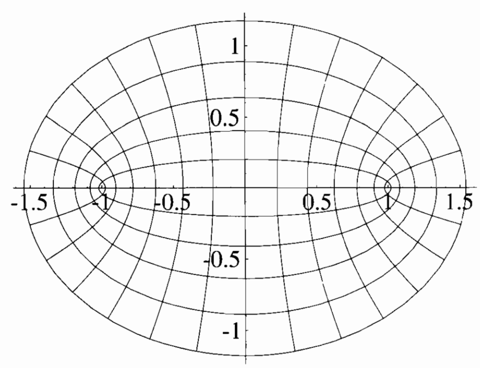
```
In[5]:= vlines = Table[N[cz], {x, -Pi/2, Pi/2, Pi/14}];
```

This command generates the graphics.

```
In[6]:= vg = ParametricPlot[ Evaluate[vlines],
                {y, -1, 1}, DisplayFunction->Identity
              ][[1]];
```

Now we combine the two sets of lines to create a picture of the Cartesian coordinate lines under the sine function.

```
In[7]:= Show[ Graphics[Join[hg, vg]],
              AspectRatio->Automatic, Axes->True ];
```



■ 1.1.3 A Simple Procedure for Drawing the Pictures

If we want to draw pictures of the coordinate lines under several functions, it becomes worthwhile to collect the necessary commands in a procedure, so that we do not have to enter them every time. The variable parts of the computation—that is, the name of the function and the ranges of the coordinates—are defined as parameters of the procedure. The commands for drawing the two sets of lines (the horizontal and the vertical lines) are so similar that we write an auxiliary procedure `Curves[]` for them. This first version CartesianMap1.m is shown in Listing 1.1–1.

```
CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_} ] :=
    Module[ {xy, x, y, hg, vg},
        xy = func[x + I y];
        hg = Curves[ xy, {x, x0, x1, dx}, {y, y0, y1} ];
        vg = Curves[ xy, {y, y0, y1, dy}, {x, x0, x1} ];
        Show[ Graphics[ Join[hg, vg] ],
              AspectRatio->Automatic, Axes->True ]
    ]

Curves[ xy_, spread_, bounds_ ] :=
    With[{curves = Table[{Re[xy], Im[xy]}, spread]},
        ParametricPlot[curves, bounds, DisplayFunction->Identity][[1]]
    ]
```

Listing 1.1–1: The first version CartesianMap1.m

- All variables local to `CartesianMap[]` are declared in a `Module[]` to protect them from any global use.
- The first argument of `CartesianMap[]` is the *name* of the function to plot. It is used in the expression `func[x + I y]`. If we want to plot a function not built into *Mathematica*, we can either define it beforehand or use a pure function (see Section 5.2).
- Observe the form of the patterns for the two ranges. Each one is a list of exactly three elements.
- In the auxiliary procedure `Curves`, the first range (`spread`) generates the distinct curves in a table; therefore, it must contain an explicit increment `dx` or `dy`. The second range (`bounds`) is used in the parametric plot and needs no increment. The number of intermediate points is determined so that the resulting curve looks smooth.
- The construction `With[{curves =` *val*`},` *expr*`]` in the procedure `Curves[]` defines a local constant `curves`. A local constant is similar to a local variable (declared with `Module[]`). Because we do not need to change its value later on, we use a constant instead of a variable. In this way, we need not wrap the argument of `ParametricPlot` in `Evaluate` either.

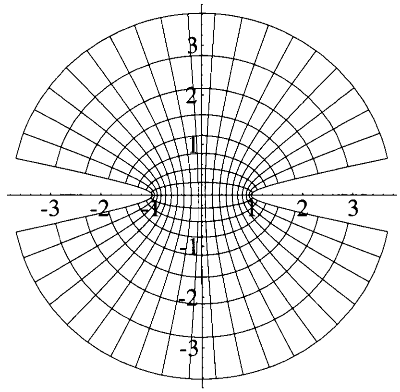
Here is an example of the use of this definition. We want to draw a picture of the cosine function.

This command reads in the definition from the file CartesianMap1.m.

```
In[1]:= << CartesianMap1.m
```

Now, we generate a picture with 20×17 lines. The lines are not closed up, because we chose a range slightly smaller than π. The picture is similar to the one of the sine function on page 5. The two functions are closely related in the complex plane.

```
In[2]:= CartesianMap[ Cos,
          {0.2, Pi-0.2, (Pi-0.4)/19}, {-2, 2, 4/16} ];
```



1.2 Basic Ingredients of a Package

In Section 1.1 we wrote a file containing our first version of the function `CartesianMap[]`. Although this file is already useful, it is not yet in a state to be published or made available to other users.

The goal in writing a *package* is to make the functions defined therein behave as much like built-in functions as possible. They should have documentation, accessible by typing `?CartesianMap`, and their behavior should not depend on the previous calculations that you have done in your *Mathematica* session before reading in the package. Several things can go wrong if you read a set of definitions into your current session:

- You could have defined values for variables that are used inside the definitions.
- You could pass variables as arguments that are also used locally inside the function.
- A function with the same name could already have been defined somewhere else.
- Auxiliary functions or private variables that are used inside the package would be accessible to the user. Users could rely on details of the implementation or make changes to it.

The following "package" `BadExample.m` illustrates one of these problems. The more subtle ones become apparent only in the context of a longer session with *Mathematica* and are then normally hard to find.

```
(* this function returns the sum of the first n powers of x *)

PowerSum[x_, n_] := Sum[ x^i, {i, 1, n} ]
```

BadExample.m: An example of bad programming style

We read in the file.

```
In[1]:= << BadExample.m
```

No problem so far. The output is as expected.

```
In[2]:= PowerSum[x, 5]

                 2    3    4    5
Out[2]= x + x  + x  + x  + x
```

The variable `i` is captured by the variable in the range of the summation. Instead of the expected `i + i^2 + i^3 + i^4 + i^5`, we get a number.

```
In[3]:= PowerSum[i, 5]

Out[3]= 3413
```

1.2.1 Isolating Local Variables

As a first step toward good programming style, let us isolate the local variable `i`, used as the summation variable, in a `Module[]`. This localization avoids the problem above, because the local symbol used is always a new one and cannot possibly conflict with anything passed

as a parameter of `PowerSum[]`. `PowerSum[]` should now do the expected thing even if the variable we give as parameter happens to be `i`. If you try this example for yourself, be sure to start a fresh *Mathematica* at this point; otherwise the previous definition would get in the way.

```
PowerSum::usage = "PowerSum[x, n] returns the sum of the first n powers of x."

PowerSum[x_, n_] :=
    Module[{i},
        Sum[ x^i, {i, 1, n} ]
    ]
```

BetterExample.m: Declaring local variables in a module

We read the file into a new *Mathematica* session.

```
In[1]:= << BetterExample.m
```

The variable `i` is not captured by the variable in the range of the summation. The output is as it should be.

```
In[2]:= PowerSum[i, 5]

               2    3    4    5
Out[2]= i + i  + i  + i  + i
```

A slight problem remains because the auxiliary symbols `i`, `x`, and `n` are visible outside the function. This will cause no harm, but could lead to confusion.

```
In[3]:= ?Global`*

i          i$         n          PowerSum x
```

■ 1.2.2 Putting Things in Their Proper Context

The mechanism that *Mathematica* provides for keeping the variables used in a package different from those used in the main session is called *contexts*. As each symbol is read from the terminal or from a file, *Mathematica* checks to see whether this symbol has already been used before. If it has been encountered before, the new instance is made to refer to that previously read symbol. Otherwise you could not refer to the value of a variable you had just defined. If the symbol has not been encountered before, a new entry in the symbol table is created.

Each symbol belongs to a certain context. Within one context the names of symbols are unique, but the same name can occur in two different contexts. By default all new symbols that you define are put in the context `Global``. The local variable `i` used in the definition of the function `PowerSum` above is therefore also in this context. (Note that context names always end with a ```.) We shall discuss contexts in greater detail in Chapter 2.

If we tell *Mathematica* to create new symbols in a different context, we can avoid the problem. The local variable `i` is now created in the context `Private``, which is not searched when you type in a variable name later on. The usage message defined for the symbol `PowerSum` is there not just to provide documentation for the function (which would be reason enough), but to make sure that the symbol `PowerSum` is defined in the current (global) context. If it had not been defined before entering the context `Private`` it, too, would not be found later on.

```
PowerSum::usage = "PowerSum[x, n] returns the sum of the first n powers of x."

Begin["Private`"]

PowerSum[x_, n_] :=
    Module[{i},
        Sum[ x^i, {i, 1, n} ]
    ]

End[]
```

BestExample.m: Isolating auxiliary symbols in a separate context

The value returned is the value of the command End[], which returns the name of the previous context.

```
In[1]:= << BestExample.m
Out[1]= Private`
```

Only the symbol PowerSum has been created in the global context.

```
In[2]:= ?Global`*
PowerSum[x, n] returns the sum of the first n powers of x.
```

All other symbols are hidden in the context Private`.

```
In[3]:= ?Private`*
Private`i  Private`i$ Private`n  Private`x
```

■ 1.2.3 A Package Context for CartesianMap

In addition to hiding local variables and functions, we also want to put all the functions that the package provides into a separate context. This context, however, must be visible so we can use the functions later on. This is achieved by the pair of commands BeginPackage[] and EndPackage[]. With these additions we present our second version CartesianMap2.m (Listing 1.2–1). The part between BeginPackage["CartMap`"] and Begin["`Private`"] is the *interface part*. It declares the functions *exported* by this package—that is, the functions that the package provides. The best way to declare a function is to give it a usage message—that is, a documentation for the function. The argument of BeginPackage[] is the context for the functions in the package.

The part of the package between Begin["`Private`"] and End[] is the *implementation part*. Here, the already-declared functions are implemented. The implementation part uses its own private context. The use of a separate context prevents details of the implementation from being exported: The implementation is *encapsulated*.

Note the initial ` in the context name inside the command Begin["`Private`"]. This establishes `Private` as a subcontext of the context CartesianMap` (so its full name is CartesianMap`Private`).

We do not get an output line from reading in the file because EndPackage[] does not return a value.

```
In[1]:= << CartesianMap2.m
```

The function CartesianMap[] is in its own context.

```
In[2]:= Context[CartesianMap]
Out[2]= CartesianMap`
```

This context, however, is accessible because it has been put on the context search path.

```
In[3]:= $ContextPath
Out[3]= {CartesianMap`,
  ProgrammingInMathematica`Options`, Global`, System`}
```

```
BeginPackage["CartesianMap`"]

CartesianMap::usage =
    "CartesianMap[f, {x0, x1, dx}, {y0, y1, dy}] plots the image
    of the Cartesian coordinate lines under the function f."

Begin["`Private`"]

CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_} ] :=
    Module[ {xy, x, y, hg, vg},
        xy = func[x + I y];
        hg = Curves[ xy, {x, x0, x1, dx}, {y, y0, y1} ];
        vg = Curves[ xy, {y, y0, y1, dy}, {x, x0, x1} ];
        Show[ Graphics[ Join[hg, vg] ],
              AspectRatio->Automatic, Axes->True ]
    ]

Curves[ xy_, spread_, bounds_ ] :=
    With[{curves = Table[{Re[xy], Im[xy]}, spread]},
        ParametricPlot[curves, bounds, DisplayFunction->Identity][[1]]
    ]

End[]

EndPackage[]
```

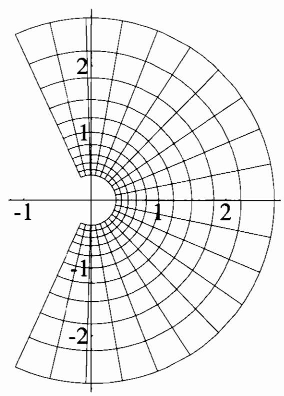
Listing 1.2–1: CartesianMap2.m: The second version of `CartesianMap[]`

Mathematica does not know auxiliary functions outside their packages.

```
In[4]:= ?Curves

Information::notfound: Symbol Curves not found.
```

The function works just as before.

```
In[5]:= CartesianMap[ Exp, {-1, 1, 0.2}, {-2, 2, 0.2} ];
```



While developing and testing a new package, it is often a good idea to leave out the `BeginPackage[]` and `EndPackage[]` calls in the early stages when the code could still contain syntax errors. If a syntax error occurs in the middle of the package, *Mathematica* may not recover completely and the contexts could end up wrong. As soon as the package is syntactically correct, the context manipulating commands can be added.

1.3 A Second Function in the Package

Instead of examining the transformation of the Cartesian coordinate lines, we can also look at the transformation of the polar coordinate lines. In polar coordinates, each point in the complex plane is described by its distance from the origin (the radius or absolute value of the complex number) and the angle of its radius line measured from the positive x-axis (the argument of the complex number). If we know the radius r and the phase angle φ of a complex number z, we can easily compute its real and imaginary parts as $r\cos\varphi$ and $r\sin\varphi$, respectively. These two formulae can be expressed more concisely as $z = re^{i\varphi}$.

First, we draw a picture of the coordinate lines. We do so in the same way as in Section 1.1, but we replace $x + iy$ by $re^{i\varphi}$.

This expression is the one for which we want to draw the lines.

```
In[1]:= z = r Exp[I p]

          I p
Out[1]= E    r
```

The coordinates of the complex number z are the real and imaginary parts.

```
In[2]:= cz = {Re[z], Im[z]}

               I p          I p
Out[2]= {Re[E    r], Im[E    r]}
```

This command generates a list of formulae for 25 lines with constant radii.

```
In[3]:= rlines = Table[ N[cz], {p, 0, 2Pi, 2Pi/24} ];
```

This command generates the graphics without plotting the image.

```
In[4]:= hg = ParametricPlot[ Evaluate[rlines],
              {r, 0, 1}, DisplayFunction->Identity
            ][[1]];
```

This command generates a list of formulae for 11 lines with constant angles.

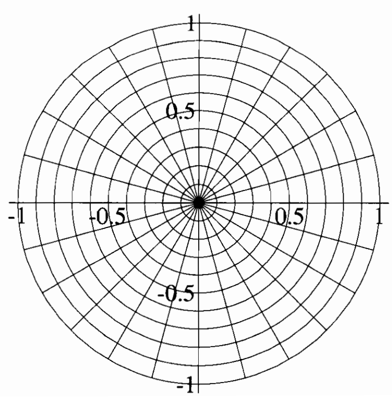
```
In[5]:= plines = Table[ N[cz], {r, 0, 1, 0.1} ];
```

This command generates the graphics without plotting the image.

```
In[6]:= vg = ParametricPlot[ Evaluate[plines],
              {p, 0, 2Pi}, DisplayFunction->Identity
            ][[1]];
```

Now, we combine the two sets of lines to create one picture of the polar coordinate lines. The lines with constant radii are circles around the origin. The lines with constant angles are rays originating in the origin.

```
In[7]:= Show[ Graphics[Join[hg, vg]],
             AspectRatio->Automatic, Axes->True ];
```



All computations are the same as in `CartesianMap[]`, except that the complex numbers are generated as `r Exp[I phi]` instead of `(x + I y)`. It is therefore useful to put the common code into an auxiliary procedure `Picture[]`. This auxiliary procedure is then used inside `CartesianMap[]` and `PolarMap[]`. It is not exported (like `Curves[]`). The two functions `CartesianMap[]` and `PolarMap[]` are defined in the same package whose name is changed to ComplexMap1.m (Listing 1.3–1). Note that we change to context name in `BeginPackage[]` to `"ProgrammingInMathematica`ComplexMap`"` to reflect the fact that all our packages are installed in a directory named ProgrammingIn-Mathematica.

```
BeginPackage["ProgrammingInMathematica`ComplexMap`"]

CartesianMap::usage =
    "CartesianMap[f, {x0, x1, dx}, {y0, y1, dy}] plots the image
    of the Cartesian coordinate lines under the function f."

PolarMap::usage =
    "PolarMap[f, {r0, r1, dr}, {p0, p1, dp}] plots the image
    of the polar coordinate lines under the function f."

Begin["`Private`"]

CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_} ] :=
    Module[ {x, y},
        Picture[ func[x + I y], {x, x0, x1, dx}, {y, y0, y1, dy} ]
    ]

PolarMap[ func_, {r0_, r1_, dr_}, {p0_, p1_, dp_} ] :=
    Module[ {r, p},
        Picture[ func[r Exp[I p]], {r, r0, r1, dr}, {p, p0, p1, dp} ]
    ]

Picture[ e_, {s_, s0_, s1_, ds_}, {t_, t0_, t1_, dt_} ] :=
    Module[ {hg, vg},
        hg = Curves[ e, {s, s0, s1, ds}, {t, t0, t1} ];
        vg = Curves[ e, {t, t0, t1, dt}, {s, s0, s1} ];
        Show[ Graphics[ Join[hg, vg] ],
              AspectRatio->Automatic, Axes->True ]
    ]

Curves[ xy_, spread_, bounds_ ] :=
    With[{curves = Table[{Re[xy], Im[xy]}, spread]},
        ParametricPlot[curves, bounds, DisplayFunction->Identity][[1]]
    ]

End[ ]

EndPackage[ ]
```

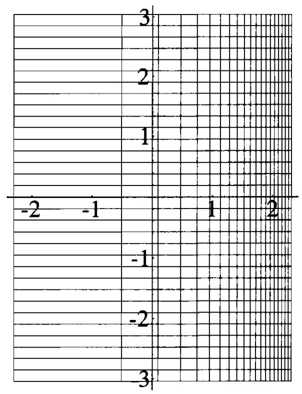
Listing 1.3–1: ComplexMap1.m: Polar and Cartesian maps in the same package

This is the preferred, system-independent way to specify a file name for input. We can use context marks to separate components and leave out the extension .m (see Section 2.5.1).

```
In[8]:= << ProgrammingInMathematica`ComplexMap1`
```

Here is a polar map of the logarithm, the inverse of the exponential function on page 11.

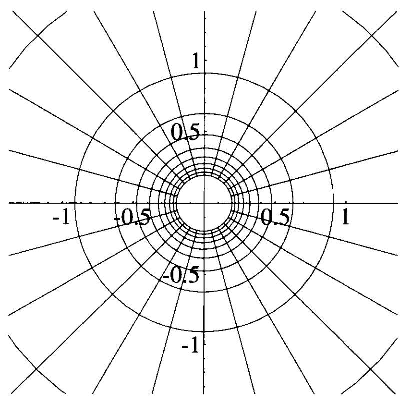
```
In[9]:= PolarMap[Log, {0.1, 10.1, 0.5},
                 {-(Pi-0.1), Pi-0.1, (Pi-0.1)/16}];
```



Here is a picture of the inversion at the unit circle. This function is not built-in, so we use a pure function instead. The formula for the inversion is $f(z) = 1/\bar{z}$ which is written as `Function[z, 1/Conjugate[z]]` in functional notation.

The images of the circles around the origin are again circles and the images of the radius lines are again radius lines. The spacing of the lines is no longer even, however (compare this with the image of the polar coordinate lines themselves earlier in this section). Note that we do not see all of the intersection points because some of them are rather far away from the origin and *Mathematica* automatically cuts off such points. The image of the origin is at infinity. Therefore, we chose to begin the radius not at 0, but at the small positive value 0.1.

```
In[10]:= PolarMap[ Function[z, 1/Conjugate[z]],
                   {0.1, 5.1, 0.5}, {-Pi, Pi, 2Pi/24} ];
```



1.4 Options

In CartesianMap[] and PolarMap[], we may want to make the increments of the two iterators optional. These increments specify the number of lines to draw. The iterators for the two variables are given as lists of three elements {*start*, *final*, *increment*}. It would be convenient if the increment had a default value as it does in other iterator constructs.

The default value should not be a constant, however; rather, we want to define the number of lines to draw and compute the increment accordingly. If we want to draw n lines between the values *start* and *final*, we compute the increment by dividing the difference of *start* and *final* by the number of lines to draw minus one (as an extra line is drawn at the end). So the formula is $(\mathit{final} - \mathit{start})/(n-1)$. What should the default value for n be?

The recommended technique is to use an *option* to specify the number of lines to draw. We shall define an option Lines for CartesianMap and PolarMap.

1.4.1 Options for Commands

An option setting is syntactically the same as a rule—for example, Lines->n. The list of all options of a function is defined as the value of Options[f]. To establish a default value of 15 for the option Lines, for example, the following definition can be used:

```
Options[CartesianMap] = { Lines -> 15 }.
```

The default is changed to a new value n by

```
SetOptions[ CartesianMap, Lines -> n ].
```

A default is overridden in a particular call of CartesianMap by giving a new setting on the command line, as in

```
CartesianMap[ ..., Lines -> n ].
```

To allow for such option settings, we need to give the pattern opts___ at the end of the arguments in the definition of CartesianMap. It is matched by any sequence of expressions, including none.

A setting in the argument list should take precedence over the global setting of Lines. Therefore, we extract the desired value of the option inside the body of the function with a replacement like this:

```
Lines /. {opts} /. Options[CartesianMap].
```

This works because the first occurrence of a rule is used in the replacement.

The first rule is used; the default does not apply in this case.

```
In[1]:= Lines /. {Lines -> 20} /. Options[CartesianMap]
Out[1]= 20
```

If no option is given on the command line, opts is the empty sequence and no replacements are done. In this case, the default in Options[CartesianMap] takes effect.

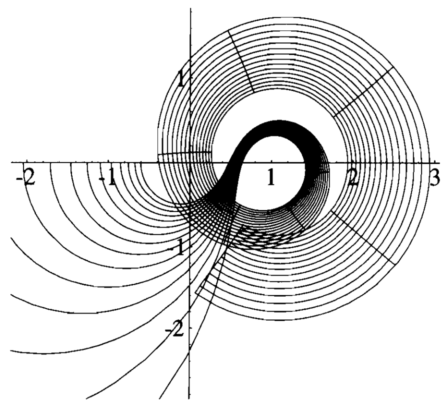
```
In[2]:= Lines /. {} /. Options[CartesianMap]
Out[2]= 15
```

The increment to use in the iterators is then the quotient of the total range by the number of lines (minus 1). The value of the option can either be a single number whose value is used for both iterators or a list of two numbers for the two iterators separately.

The new program is in ComplexMap2.m, shown partially in Listing 1.4–1. Note that there are two definitions for each command; the first definition is used if the increments in the iterators are present. The second definition is used if the increments are missing. The increment is then computed in the manner just described and the command is called again—this time with the increments.

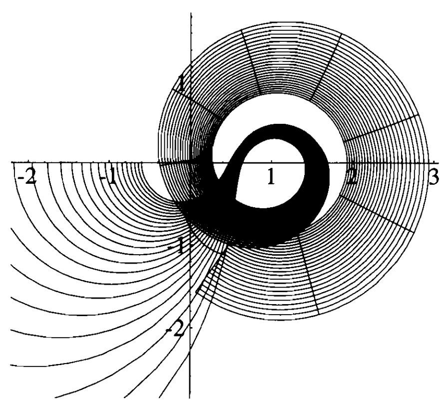
The Riemann zeta function with a default of 15 lines each.

```
In[1]:= CartesianMap[ Zeta, {0.1, 0.9}, {0, 20} ];
```



We can specify a different number of lines in this picture by setting the option Lines.

```
In[2]:= CartesianMap[ Zeta, {0.1, 0.9}, {0, 20},
                      Lines -> 25 ];
```



To change the default, we can reset the option Lines to a new value.

```
In[3]:= SetOptions[ PolarMap, Lines -> 25 ]
Out[3]= {Lines -> 25}
```

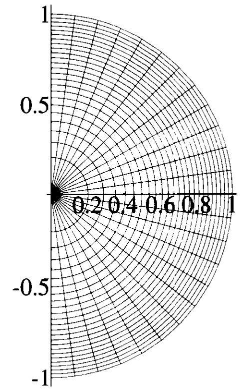
```
BeginPackage["ProgrammingInMathematica`ComplexMap`"]

CartesianMap::usage = "CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}, opts..]
    plots the image of the cartesian coordinate lines under the function f.
    The default values of dx and dy are chosen so that the number of lines
    is equal to the value of the option Lines."
⋮

Lines::usage = "Lines->{lx, ly} is an option of CartesianMap and PolarMap
    that gives the number of lines to draw."

Begin["`Private`"]

Options[CartesianMap] = {Lines->15}
Options[PolarMap] = {Lines->15}

(* explicit increments *)

CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_}, opts___ ] :=
    Module[ {x, y},
        Picture[ func[x + I y], {x, x0, x1, dx}, {y, y0, y1, dy} ] ]
PolarMap[ func_, {r0_, r1_, dr_}, {p0_, p1_, dp_}, opts___ ] :=
    Module[ {r, p},
        Picture[ func[r Exp[I p]], {r, r0, r1, dr}, {p, p0, p1, dp} ] ]

(* default increments *)

CartesianMap[ func_, {x0_, x1_}, {y0_, y1_}, opts___ ] :=
    Module[ {lines, dx, dy},
        lines = Lines /. {opts} /. Options[CartesianMap];
        If[ Head[lines] =!= List, lines = {lines, lines} ];
        dx = N[(x1 - x0)/(lines[[1]]-1)];
        dy = N[(y1 - y0)/(lines[[2]]-1)];
        CartesianMap[ func, {x0, x1, dx}, {y0, y1, dy}, opts ] ]

PolarMap[ func_, {r0_, r1_}, {p0_, p1_}, opts___ ] :=
    Module[ {lines, dr, dp},
        lines = Lines /. {opts} /. Options[PolarMap];
        If[ Head[lines] =!= List, lines = {lines, lines} ];
        dr = N[(r1 - r0)/(lines[[1]]-1)];
        dp = N[(p1 - p0)/(lines[[2]]-1)];
        PolarMap[ func, {r0, r1, dr}, {p0, p1, dp}, opts ] ]
⋮

End[ ]
EndPackage[ ]
```

Listing 1.4–1: Part of ComplexMap2.m: Default values

This new default value is now used without the need to give the option in the command.

The square root function halves the angle of each complex number. The full circle from $-\pi$ to π is therefore mapped into half a circle.

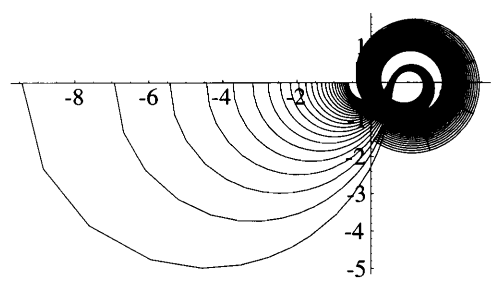
```
In[4]:= PolarMap[ Sqrt, {0, 1}, {-Pi, Pi} ];
```



The example with the zeta function shows that points that lie far away from the origin are clipped automatically. This is usually a desirable feature. We can, however, change the range of values plotted with the `PlotRange` option.

This picture shows more of the lines on the left, but details are lost overall.

```
In[5]:= Show[ %2, PlotRange -> All ];
```



■ 1.4.2 Advanced Topic: Passing on Options

Our commands `CartesianMap[]` or `PolarMap[]` eventually call the graphics functions `ParametricPlot[]` and `Show[]`. These graphics functions have many options that we may want to change. To make such changes possible, we should pass on all options given in a call of `CartesianMap[]` and `PolarMap[]`. But we have to be careful to pass on only those options that are valid for the graphics function (and not our own option `Lines`, for example).

The auxiliary function `FilterOptions[`*cmd*`, `*opts*`...]` selects from a sequence of options those that are valid for the command *cmd*. It is defined in the standard package Utilities/FilterOptions.m and discussed in Section 3.2.4. To use `FilterOptions`, we have to *import* the package into our own package. We do so with the command ``Needs["Utilities`FilterOptions`"]`` at the beginning of the implementation part of

our package, right after `Begin["`Private`"]`. The imported package is then read in, and its functions can be used in the implementation part of our own package. This new version of our package is ComplexMap3.m. Listing 1.4–2 shows the code for `Picture[]` and `Curves[]`. Note that the changes affect only the auxiliary functions; the advantages of putting common code into auxiliary functions become once more apparent.

```
⋮
Begin["`Private`"]

Needs["Utilities`FilterOptions`"]

Options[CartesianMap] = Options[PolarMap] = {Lines->15}

CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_}, opts___ ] :=
    Module[ {x, y},
        Picture[ func[x + I y], {x, x0, x1, dx}, {y, y0, y1, dy}, opts ] ]

PolarMap[ func_, {r0_, r1_, dr_}, {p0_, p1_, dp_}, opts___ ] :=
    Module[ {r, p},
        Picture[ func[r Exp[I p]], {r, r0, r1, dr}, {p, p0, p1, dp} ] ]
⋮
Picture[ e_, {s_, s0_, s1_, ds_}, {t_, t0_, t1_, dt_}, opts___ ] :=
    Module[ {hg, vg},
        hg = Curves[ e, {s, s0, s1, ds}, {t, t0, t1}, opts ];
        vg = Curves[ e, {t, t0, t1, dt}, {s, s0, s1}, opts ];
        Show[ Graphics[ Join[hg, vg] ],
              FilterOptions[Graphics, opts],
              AspectRatio -> Automatic, Axes -> True ] ]

Curves[ xy_, spread_, bounds_, opts___ ] :=
    With[{curves = Table[{Re[xy], Im[xy]}, spread]},
         ParametricPlot[ curves, bounds, DisplayFunction -> Identity,
                         Evaluate[FilterOptions[ParametricPlot, opts]] ][[1]]
    ]
⋮
```

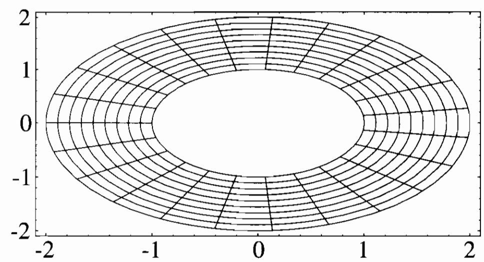
Listing 1.4–2: Part of ComplexMap3.m: Passing options to the graphics functions

Note how the options are passed down from `CartesianMap` first to `Picture` and then to `Curves`. The filtered options are inserted into the argument list of `Show` and `ParametricPlot`, respectively. The placement of these options in the argument list of `Show[]` is significant. They come before the options `AspectRatio` and `Axes`. This placement allows us to override these values because the first encounter of an option is used. If we had put `opts` at the end, there would have been no way to change the settings for `AspectRatio` or `Axes`.

`ParametricPlot` has the attribute `HoldAll`, which prevents the correct evaluation of the argument `FilterOptions[ParametricPlot, opts]`. Therefore, we must force the evaluation by wrapping the argument in `Evaluate[]`.

Here is another picture of the identity. We give different values for `AspectRatio`, suppress axes, and draw a frame around the picture.

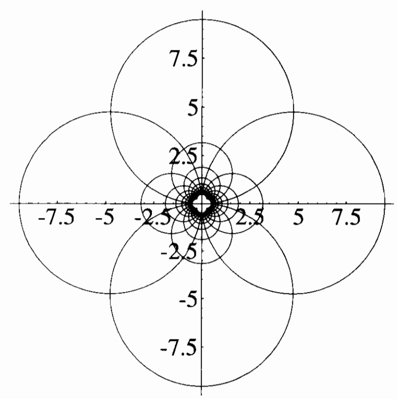
```
In[1]:= PolarMap[ Identity, {1, 2}, {-Pi, Pi},
                  Lines -> {10, 24}, AspectRatio -> 0.5,
                  Axes -> None, Frame -> True ];
```



Here is another picture of the inversion (see Section 1.3 for the first one). This time we use `CartesianMap[]` and also try to plot all points, even those far away from the center. For smoother curves, we a higher value of the option `PlotPoints`, used in `ParametricPlot`.

The images of the straight lines become circles under inversion. The coordinate lines that pass close by the origin are taken farthest away from it and do not look at all like circles there. Thus, it is normally a good idea to cut off such points as *Mathematica* does it, by default.

```
In[2]:= CartesianMap[ Function[z, 1/Conjugate[z]],
                      {-2, 2, 4/19}, {-2, 2, 4/19},
                      PlotRange -> All, PlotPoints -> 30 ];
```



1.5 Defaults for Positional Arguments

Optional arguments of a procedure are arguments that you can leave out when calling the procedure. In this case, a *default value* is used.

A (constant) default for an argument can be declared with *var*_:*default*. If an argument in a definition is declared in this way (with a default), the definition is also applied if the argument is missing. The pattern variable *var* takes on the value *default* in this case.

This rule says that the default for the second argument of `f` should be 17.

```
In[1]:= f[ x_, y_:17 ] := {x, y}
```

If the second argument is given, the default is ignored.

```
In[2]:= f[ 4, 6 ]
Out[2]= {4, 6}
```

But if the second argument is missing, the default value is used, and the rule matches even though only one argument was given.

```
In[3]:= f[ 4 ]
Out[3]= {4, 17}
```

Please note that the default value cannot depend on the other parameters (`x`, for example). It is evaluated when the rule is given, rather than later on when the rule is used. In more complicated cases, it is better to give a second rule that computes the default value and then calls the other rule.

Here is the base case. It is used if both arguments are given.

```
In[4]:= g[ x_, y_ ] := {x, y}
```

This rule is used if the second argument is missing. A value for it is then computed, and the function `g` is called again—this time, however, with two arguments.

```
In[5]:= g[ x_ ] := g[ x, 2x ]
```

The rule with only one argument plays no role if both arguments are given in a call.

```
In[6]:= g[ 4, 6 ]
Out[6]= {4, 6}
```

Now, a value is computed for the second argument.

```
In[7]:= g[ 4 ]
Out[7]= {4, 8}
```

1.5.1 Computed Defaults

In Section 1.4.1, we added rules that allow us to leave out the increments in the range specifiers for `CartesianMap[]` and `PolarMap[]`. But what if we want to leave it out in one of the ranges and explicitly give an increment for the second one? We might want to say `PolarMap[Sqrt, {0, 1, 0.2}, {-Pi, Pi}]`. There is no rule which will match, because the first rule for `CartesianMap[]` required both ranges to have three elements while the second one requires them to have two elements each (see Listing 1.4–2). We would need two more rules to cover the two mixed cases.

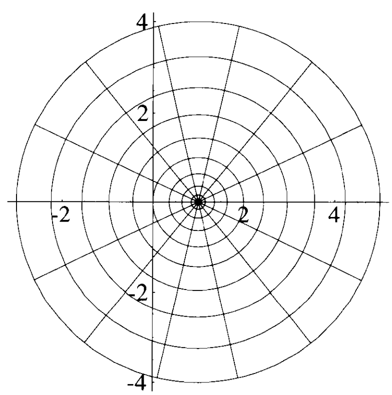
There is a way out. The only reason that we needed the extra rules was that the default values had to be computed and were not constants that could have been specified easily in the rule, as was the default value 0 for the radius in `PolarMap[]`. The idea is to use some symbol as default value for the increment and then test for the presence of this symbol. The symbol `Automatic` can be used for this purpose. It is used for similar purposes as a value of many options of graphics functions. To test whether the value of the increment is this symbol, we use `SameQ[`e_1`, `e_2`]` (or e_1===e_2), which tests equality of symbols. If the value of an increment is equal to `Automatic`, a suitable numerical value is computed; otherwise, the given value is used. This computation is done independently for each increment. The treatment of increments can now be put into the auxiliary function `Picture[]`, because it is similar for both `CartesianMap[]` and `PolarMap[]` (note that we pass the name of the command as a new first argument to allow access to its option list inside `Picture[]`). As another convenience we give `r0` the default value 0. Most of the time we want to start the radius lines at the origin. Because this default is a simple number, we can put it right into the pattern as `r0_:0`. These improvements are part of ComplexMap4.m, shown partially in Listing 1.5–1.

Names for patterns, such as `ds` and `dt` above, cannot be used as local variables inside the body of the definition. Therefore, we declare two local variables `nds` and `ndt` in a module and initialize them with the values of `ds` and `dt`.

The function $z \mapsto z^2 + 1$ maps the origin into the point $(1, 0)$ and doubles angles; therefore, the upper half plane is mapped onto the whole complex plane.

The range for r uses an explicit increment, the range for φ is determined by the default of the option `Lines`.

```
In[1]:= PolarMap[ Function[z, z^2 + 1],
                  {0, 2, 0.2}, {0, Pi} ];
```



■ 1.5.2 Defaults or Options?

Defaults are convenient because they save a lot of repetitive typing. But too many of them can be confusing. Already, we have a possible conflict in the function `PolarMap[]`. There is a default for both the start value and the increment for the first range. So what does the range specifier `{`a`, `b`}` mean? Is a the start value, b the final value, and the increment

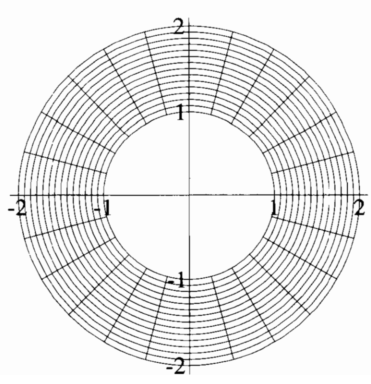
```
BeginPackage["ProgrammingInMathematica`ComplexMap`"]
⋮
Begin["`Private`"]
⋮
CartesianMap[ func_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic},
              opts___ ] :=
    Module[ {x, y}, Picture[ CartesianMap, func[x + I y],
                             {x, x0, x1, dx}, {y, y0, y1, dy}, opts ] ]

PolarMap[ func_, {r0_:0, r1_, dr_:Automatic}, {p0_, p1_, dp_:Automatic},
          opts___ ] :=
    Module[ {r, p}, Picture[ PolarMap, func[r Exp[I p]],
                             {r, r0, r1, dr}, {p, p0, p1, dp}, opts ] ]

Picture[ cmd_, e_, {s_, s0_, s1_, ds_}, {t_, t0_, t1_, dt_}, opts___ ] :=
    Module[ {hg, vg, lines, nds = ds, ndt = dt},
        lines = Lines /. {opts} /. Options[cmd];
        If[ Head[lines] =!= List, lines = {lines, lines} ];
        If[ ds === Automatic, nds = N[(s1-s0)/(lines[[1]]-1)] ];
        If[ dt === Automatic, ndt = N[(t1-t0)/(lines[[2]]-1)] ];
        hg = Curves[ e, {s, s0, s1, nds}, {t, t0, t1}, opts ];
        vg = Curves[ e, {t, t0, t1, ndt}, {s, s0, s1}, opts ];
        Show[ Graphics[ Join[hg, vg] ],
              FilterOptions[Graphics, opts],
              AspectRatio -> Automatic, Axes -> True ]
    ]
⋮
End[ ]

EndPackage[ ]
```

Listing 1.5–1: Part of ComplexMap4.m: Better treatment of defaults than in ComplexMap3.m

computed by default? Or does the start value default to 0 and is a the final value and b the increment? Matching from left to right seems more natural, and indeed this is what happens.

The identity is a good test example because it does not deform the coordinate lines. The radius lines are between 1 and 2, and their number is equal to the option `Lines`. The angular increment is equal to $\pi/12$, which gives 25 rays (the last one is superimposed on the first one, however).

```
In[2]:= PolarMap[ Identity, {1, 2}, {-Pi, Pi, Pi/12} ];
```



Arguments of a function whose meaning depends on their positions in the argument list are called *positional arguments.* The first three arguments of `CartesianMap[]` and `PolarMap[]` are positional. Because of their shortcomings, especially in interactive use, there are alternatives in some languages, the so-called *named arguments.* Named arguments are identified by their names and not by their positions in the argument list. They can be given in any order, or left out completely. In *Mathematica* such named arguments are called *options.* Their use should be considered whenever a function has many features that should be user-settable, but when in most applications a default value is adequate. Graphics functions tend to have many such arguments, as there are many aspects of a picture that you might want to change only occasionally. How to define options for your own functions is discussed further in Section 3.2.

■ 1.6 Parameter Type Checking

So far we have not paid any attention to possibly bad parameters that a user of our package might accidentally type in. If a built-in function is called with a bad parameter value, it usually prints an error message and returns itself unevaluated.

The subtle typo is caught and the input is returned.

```
In[1]:= ParametricPlot[{x, y}, {x, -1, l}, {y, -1, 1}]

ParametricPlot::plln:
   Limiting value l in {x, -1, l}
     is not a machine-size real number.

Out[1]= ParametricPlot[{x, y}, {x, -1, l}, {y, -1, 1}]
```

What happens if `CartesianMap[]` or `PolarMap[]` is called with bad parameters? If the two ranges (the second and third arguments of these functions) are given with the wrong number of elements (only one or more than three), the rules will simply not match. Things are worse if the number of arguments is correct, but one of the values inside the range does not evaluate to a number. In this case the `Table[]` command inside the function body will generate an error message. Its value is then not a list of numbers, and most of the following commands will also generate error messages. Finally, the `Show[]` command will complain that it has not received a valid graphics specification as input. If the user does not know how our function works internally (and there should be no reason for him to have to know), then these error messages will be very confusing because they are not recognized as a consequence of the original error. (Try, for example, to evaluate `CartesianMap[Log, {-pi, pi, 0.1}, {-1, 1}]` with the common mistake of writing `pi` instead of `Pi`.)

To be "user-friendly," our program should check parameter values as well as it can. One obvious condition is that all the elements in the two ranges should evaluate to numbers. Listing 1.6–1 shows `CartesianMap[]` with these checks added as conditions at the end.

```
CartesianMap[ func_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic},
              opts___?OptionQ ] :=
    Module[ {x, y}, Picture[ CartesianMap, func[x + I y],
                             {x, x0, x1, dx}, {y, y0, y1, dy}, opts ]
    ] /;
    NumericQ[x0] && NumericQ[x1] && NumericQ[y0] && NumericQ[y1] &&
    (NumericQ[dx] || dx === Automatic) && (NumericQ[dy] || dy === Automatic)
```

Listing 1.6–1: Part of ComplexMap5.m: Error checks

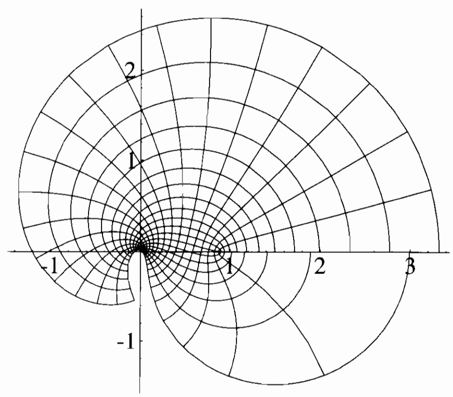
It would be too restrictive to simply test for `NumberQ[x0]`. `x0` could be a constant (like `Pi`) or a function value (like `Sin[1]`) that is not a number but that does evaluate to a number when `N[]` is applied to it. The predicate `NumericQ[]` takes such cases into account and returns `True` if its argument is numerical, perhaps after applying `N[]` to it. Note that we must allow for the increments to be the symbol `Automatic` instead of a numerical quantity.

The terms connected with the logical or (two vertical bars in *Mathematica* notation) have to be put in parentheses because the priority of `&&` is higher than is the priority of `||`.

Another common mistake is to call `CartesianMap[]` with extra arguments at the end that are not options. We shall later give a method to make sure that the options are all valid (see Section 3.2.4), but for now we just want to check that these arguments are all syntactically correct, that is, that they are rules of the form *name* `->` *value* or *name* `:>` *value* or lists of such rules. The built-in predicate `OptionQ[`*arg*`]` tests whether its argument is of this form. We change the parameter from `opts___` to `opts___?OptionQ`. This pattern is matched only by a sequence of options.

As a final picture in Chapter 1, here is a Cartesian map of the Γ function. We shall return to the package ComplexMap.m in Chapter 3.

```
In[2]:= CartesianMap[ Gamma, {0.3, 3.5}, {0, 3.6},
             Lines -> {20, 20}, PlotRange -> All ];
```



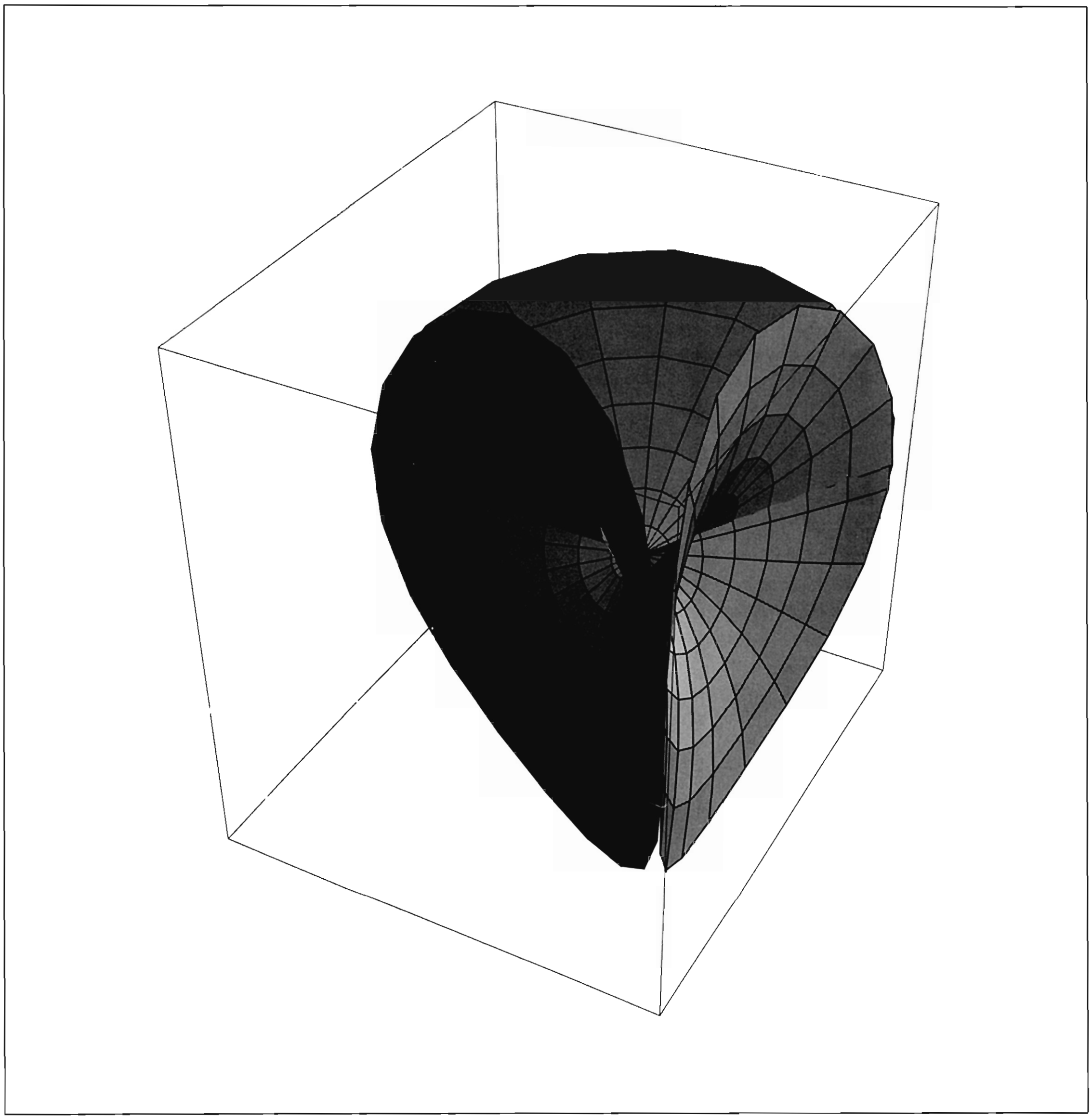
The final version of ComplexMap.m (with the extensions from Chapter 3) is shown in Listing 1.6–2. Note that a copy of this package is also available as standard package Graphics'ComplexMap'.

```
BeginPackage["ProgrammingInMathematica`ComplexMap`"]

CartesianMap::usage = "CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}] plots
    the image of the cartesian coordinate lines under the function f.
    The default values of dx and dy are chosen so that the number of lines
    is equal to the value of the option Lines."
PolarMap::usage = "PolarMap[f, {r0:0, r1, (dr)}, {phi0, phi1, (dphi)}] plots the
    image of the polar coordinate lines under the function f. The default for
    the phi range is {0, 2Pi}. The default values of dr and dphi are chosen
    so that the number of lines is equal to the value of the option Lines."
Lines::usage = "Lines -> {lx, ly} is an option of CartesianMap and PolarMap
    that gives the number of lines to draw."
$Lines::usage = "$Lines is the default of the option Lines. The value should be
    a positive integer or a list of two positive integers."

Begin["`Private`"]

Needs["Utilities`FilterOptions`"]

$Lines = 15; (* global default *)
Options[CartesianMap] = Options[PolarMap] = { Lines :> $Lines }

CartesianMap[ func_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic},
              opts___?OptionQ ] :=
    Module[ {x, y}, Picture[ CartesianMap, func[x + I y],
                             {x, x0, x1, dx}, {y, y0, y1, dy}, opts ]
    ] /; NumericQ[x0] && NumericQ[x1] && NumericQ[y0] && NumericQ[y1] &&
     (NumericQ[dx] || dx === Automatic) && (NumericQ[dy] || dy === Automatic)

PolarMap[ func_, {r0_:0, r1_, dr_:Automatic}, {p0_, p1_, dp_:Automatic},
          opts___?OptionQ ] :=
    Module[ {r, p}, Picture[ PolarMap, func[r Exp[I p]],
                             {r, r0, r1, dr}, {p, p0, p1, dp}, opts ]
    ] /; NumericQ[r0] && NumericQ[r1] && NumericQ[p0] && NumericQ[p1] &&
     (NumericQ[dr] || dr === Automatic) && (NumericQ[dp] || dp === Automatic)

PolarMap[ func_, rr_List, opts___?OptionQ ] := PolarMap[ func, rr, {0, 2Pi}, opts ]

Picture[ cmd_, e_, {s_, s0_, s1_, ds_}, {t_, t0_, t1_, dt_}, opts___ ] :=
    Module[ {hg, vg, lines, nds = ds, ndt = dt},
        lines = Lines /. {opts} /. Options[cmd];
        If[ Head[lines] =!= List, lines = {lines, lines} ];
        If[ ds === Automatic, nds = N[(s1-s0)/(lines[[1]]-1)] ];
        If[ dt === Automatic, ndt = N[(t1-t0)/(lines[[2]]-1)] ];
        hg = Curves[ e, {s, s0, s1, nds}, {t, t0, t1}, opts ];
        vg = Curves[ e, {t, t0, t1, ndt}, {s, s0, s1}, opts ];
        Show[ Graphics[ Join[hg, vg] ], FilterOptions[Graphics, opts],
              AspectRatio -> Automatic, Axes -> True ] ]

Curves[ xy_, spread_, bounds_, opts___ ] :=
    With[{curves = Table[{Re[xy], Im[xy]}, spread]},
        ParametricPlot[ curves, bounds, DisplayFunction -> Identity,
                        Evaluate[FilterOptions[ParametricPlot, opts]] ][[1]] ]

End[ ]
Protect[ CartesianMap, PolarMap, Lines ]
EndPackage[ ]
```

Listing 1.6–2: ComplexMap.m

Chapter 2

Packages



In Chapter 1 we used some of the tools that *Mathematica* provides to write a package. Now we want to look in detail at all the facilities involving packages.

The first important concept is that of a *context*. Contexts provide a way of keeping variables in different packages separate. Another benefit of contexts is the ability to hide information from the user of our package. We want to keep local variables and auxiliary functions hidden from the user. Contexts are discussed in detail in Section 1.

Section 2 introduces the concept of importing another package into a package. This allows us to use another package inside our own or to add to an already existing package.

In Section 3 we present the tools for protecting a package against inadvertent modification by users. This is done in the same way as for built-in objects.

Next we present a *skeletal package*. It contains all the commands for setting up the proper contexts and can be used as a starting point for your own packages. Using such a template guarantees a certain uniformity in the overall appearance of a package, making it easier to understand packages that somebody else has written. We shall also discuss documentation tools that allow for the automatic classification and indexing of programs.

In Section 5 we discuss the correspondence between context names (meaningful to *Mathematica*) and file names (meaningful to the operating system) and treat a particular problem that arises if functions in a package are referenced before the package has been loaded. Packages can also be set up to be loaded on demand, whenever a function from them is used. We present the tools needed to set up your own packages for loading on demand.

Finally, Section 6 discusses the issues involved in designing a larger application that consists of several packages.

About the illustration overleaf:
A minimal surface.This one was generated as a parametric surface with coordinates

$$(r\cos\varphi - \frac{r^2}{2}\cos 2\varphi, -r\sin\varphi - \frac{r^2}{2}\sin 2\varphi, \frac{4}{3}r^{3/2}\cos\frac{3}{2}\varphi)\,.$$

Minimal surfaces have interesting mathematical properties. They are not easy to imagine given only their formulae. A picture helps a lot.

2.1 Contexts

Contexts (or name spaces) provide a way to keep variables in different packages separate, even if their names are the same. Contexts also allow us to hide unimportant (internal) symbols.

2.1.1 Contexts of Symbols

Recall from Section 1.2 that each symbol belongs to a context. There are two global variables in *Mathematica* that control the creation of new symbols and the lookup of existing ones.

`$Context`	the current context
`$ContextPath`	the list of contexts to search

These two variables govern the lookup of contexts

When *Mathematica* encounters a symbol in the input that you type or that is read from a file, it searches the current context and then all the contexts on the context path for this symbol. If it cannot find one, a new symbol is created in the current context. See also Sections 2.6.8–2.6.10 of the *Mathematica* book for additional explanations on contexts.

2.1.2 Contexts in Packages

Normally, the values of the variables `$Context` or `$ContextPath` should not be changed directly. It is better to use the commands provided for context manipulation. They make the necessary changes to these variables and perform some error checking.

`BeginPackage["`*Context*`` ` ``"]`	start a package
`EndPackage[ ]`	end a package
`Begin["`*Context*`` ` ``"]`	change the current context
`End[ ]`	return to the previous context

Commands to manipulate contexts

To see how these commands change the values of `$Context` and `$ContextPath`, let us look in detail at what happens when a package is read in. We simulate reading in a stripped-down version of the example from Chapter 1. We take out all the actual code and are left with just a few declarations:

```
BeginPackage["ComplexMap`"]

CartesianMap::usage = "CartesianMap[f,...] plots a map."

Begin["`Private`"]

CartesianMap[ func_,... ] := ...

Picture[ e_, ... ] :=
    Module[ {hg, ...}, ...]

End[ ]
EndPackage[ ]
```

An excerpt from ComplexMap.m

In Table 2.1–1, we show the effect of each line in the package on the values of the variables `$Context` and `$ContextPath`. The entries are filled in only if the command changed them. We also show the fully qualified names of the symbols encountered in the input. Note that we use the simple context name `ComplexMap`` in this example, rather than the nested name `ProgrammingInMathematica`ComplexMap`` as it appears effectively in the file. We do this to keep things simpler and to save some space in the wide tables that follow.

The first line is not part of the package. It simulates some computations that we did before reading in the package. At the end, we return to this global context and use the function `CartesianMap[]` that we have just defined. Ordinarily, the package is read in by `<<ComplexMap`` or by `Needs["ComplexMap`"]` in one step and we do not see the effects of the individual lines as we do here. It is, however, possible to enter the lines of a package line by line, as we do here, for debugging purposes.

Here is a line-by-line discussion of what happens exactly in the example above:

1. This line shows a typical calculation that could have been done just before reading in the package ComplexMap.m. The values of the variables `$Context` and `$ContextPath` are the default ones when *Mathematica* starts up.

2. Here we give the command to read in the package. The following lines are the contents of the package; they are not typed in by the user.

3. `BeginPackage["`*PackageContext*`` `"]` sets the value of `$Context` to its argument *PackageContext*``. `$ContextPath` always becomes `{`*PackageContext*``, `System``}` independent of what it was before. Note especially that `Global`` is not on the context search path. In a package there is, therefore, no danger of wrongly accessing any objects that have been defined in the *Mathematica* session so far. The package always starts in a clean state.

4. Defining a usage message for `CartesianMap` creates the symbol because neither is it built in nor does it exist in the current context. It is created in the current context, which is the package context `ComplexMap``.

5. The command `Begin["`Private`"]` changes the current context. It does not affect the context path. The argument ``"`Private`"`` begins with a context mark ` and the context is, therefore, a *subcontext* of the current context, with full name

	Command	Symbols	$Context	$ContextPath
1	`Factor[y^2-1]`	`` System`Factor, Global`y ``	`` Global` ``	`` {Global`, System`} ``
2	`<<ComplexMap.m`			
the following lines are read from the package				
3	`` BeginPackage["ComplexMap`"] ``	`` System`BeginPackage ``	`` ComplexMap` ``	`` {ComplexMap`, System`} ``
4	`CartesianMap::usage = "..."`	`` ComplexMap`CartesianMap ``		
5	`` Begin["`Private`"] ``	`` System`Begin ``	`` ComplexMap`Private` ``	
6	`CartesianMap[ func_, ...] :=`	`` ComplexMap`CartesianMap ``		
		`` ComplexMap`Private`func ``		
7	`Picture[ e_, ...] :=`	`` ComplexMap`Private`Picture ``		
		`` ComplexMap`Private`e ``		
8	`Module[{hg, ...}, ...]`	`` ComplexMap`Private`hg ``		
9	`End[ ]`		`` ComplexMap` ``	
10	`EndPackage[ ]`		`` Global` ``	`` {ComplexMap`, Global`, System`} ``
here we return to the top level				
11	`CartesianMap[Sin, ...]`	`` ComplexMap`CartesianMap ``		
		`` System`Sin ``		
12	`Picture[...]`	`` Global`Picture ``		

Table 2.1–1: Context manipulations during the reading of a package.

`` ComplexMap`Private` ``. The consequence is that newly created symbols will be put in the context `` ComplexMap`Private` ``.

6. Here is the definition of the function `CartesianMap[]`. The symbol `CartesianMap` has already been created (in line 4) in the context `` ComplexMap` ``. Because this context is on the context path the symbol is found there and the definition is for the existing symbol.

7. The auxiliary procedure `Picture[]` is local to the package. The symbol `Picture` is created in the current context `` ComplexMap`Private` ``. This context will no longer be accessible after the package has been read in. The pattern variable `e` is also hidden in this private context.

8. The variable `lines` is local to the command `Picture[]`. (More explanations about `Module[]` are given in Section 5.6.1.)

9. The command `End[]` undoes the previous `Begin[]` and restores the current context to `` ComplexMap` ``. The context `` ComplexMap`Private` `` is not on the context path, and any symbols defined in this context are therefore no longer accessible.

10. `EndPackage[]` restores the current context to what it was before the command `BeginPackage[]`. The context path is also restored, but the new package context is added in front of it.

11. Now we are back in our interactive *Mathematica* session and can use the command `CartesianMap[]` because the context in which it was defined appears in the context path.

12. The command `Picture[]`, however, cannot be used. The symbol is not found in its context `ComplexMap`Private``. A new symbol is created in the global context that has nothing to do with the other one.

The net effect of reading in a package is to add a new context in front of the context path and to define some functions in this context. Functions that are made available in a self-contained program unit for use outside it are said to be *exported* from it. The package controls which of its functions it wants to export. This mechanism of explicitly exporting objects from a program unit is found in one form or another in most modern programming languages. It is an important software engineering tool.

Context names themselves are not symbols and must be quoted when used as arguments of a command. A symbol in any context can be specified with its fully qualified name as *Context`symbol*. It is, therefore, not entirely true that the auxiliary function `Picture[]` is inaccessible. By using `ComplexMap`Private`Picture`, we could access it even though the context in which it is defined is not on the search path. This practice is, of course, strongly discouraged.

■ 2.1.3 Tiny Packages

If we write code for one or two small commands that do not have any auxiliary functions, it is probably not worth creating a full-blown package. Nevertheless, it is a good idea to put the definitions into a separate, private context to avoid the kind of problems outlined at the beginning of Section 1.2. Such a mini-package will look like this:

```
ExpandBoth::usage = "ExpandBoth[e] expands all numerators and denominators in e."

Begin["`Private`"]

ExpandBoth[x_Plus] := ExpandBoth /@ x
ExpandBoth[x_] := Expand[ Numerator[x] ] / Expand[ Denominator[x] ]

End[]
Null
```

ExpandBoth.m: A tiny package

The context used for the implementation of `ExpandBoth[]` is specified as a subcontext of the current context—whatever it is at the time the tiny package is read. Commands like this can be put into the initialization file init.m and are then available in every *Mathematica* session you start.

```
In[1]:= << ProgrammingInMathematica/ExpandBoth.m
```

Here is an expression with numerators and denominators that you might want to expand separately.

```
In[2]:= (1 + Sqrt[5])^2/3 + (a + b)^2/(e - I)^3

                      2          2
        (1 + Sqrt[5])     (a + b)
Out[2]= -------------- + ---------
              3                  3
                         (-I + e)
```

Each numerator and denominator is expanded.

```
In[3]:= ExpandBoth[ % ]

                           2              2
        6 + 2 Sqrt[5]     a  + 2 a b + b
Out[3]= ------------- + --------------------
              3                      2    3
                        I - 3 e - 3 I e  + e
```

Compare this with the built-in ExpandAll[] that puts each term of the numerator over a separate copy of the denominator.

```
In[4]:= ExpandAll[ %% ]

                                       2
            2 Sqrt[5]                 a
Out[4]= 2 + --------- + -------------------- +
                3                    2    3
                        I - 3 e - 3 I e  + e

                                     2
         2 a b                      b
  -------------------- + --------------------
                2    3                2    3
  I - 3 e - 3 I e  + e   I - 3 e - 3 I e  + e
```

2.2 Packages That Use Other Packages

Recall from Section 2.1 that the command `BeginPackage["`*Context*`` ` ``"]` resets the context path to `{`*Context*`` ` ``, System`}`, independent of its former value. One consequence is that symbols that the user of your package has defined before reading in the package do not get in the way because such symbols are defined in the global context. But it also means that any other packages that have been read in are not accessible inside your package. You might want to use a command from another package inside your own package, however.

2.2.1 Importing Another Package

`BeginPackage[]` has optional arguments that specify contexts that are to be left on the search path. If they are not already on the search path they are first read in using the command `Needs["`*Context*`` ` ``"]` implicitly. An example of such a package is the standard package Graphics‘Shapes‘. It defines a command to rotate a three-dimensional graphic object and uses the standard package Geometry‘Rotations‘ to compute the rotation matrix. Listing 2.2–1 gives a small excerpt showing how this is set up. The version of this package shown here is an old one that was distributed with *Mathematica* Version 2.0. The newer one will be described in the next subsection.

```
BeginPackage["Graphics`Shapes`", "Geometry`Rotations`"]

RotateShape::usage = "..."

Begin["`Private`"]

RotateShape[ shape_, phi_, theta_, psi_ ] :=
    Module[{rotmat = RotationMatrix3D[N[phi], N[theta], N[psi]]},
              ⋮
    ]

End[ ]

EndPackage[ ]
```

Listing 2.2–1: Part of standard package Graphics/Shapes.m, old version

The function `RotationMatrix3D[]` used inside `RotateShape[]` comes from the package Geometry/Rotations.m. Table 2.2–1 shows the detailed analysis of the context changes in the same format as in Section 2.1, Table 2.1–1. To save some space in the table, we used the context `` Shapes` `` instead of `` Graphics`Shapes` ``.

As with the command `Needs[]` itself, no package is read in if the context given as an optional argument to `BeginPackage[]` is already on the search path. A package is, therefore, read in only once even if it is used inside several packages that you read in your session one after the other.

	Command	Symbols	`$Context`	`$ContextPath`
1	`BeginPackage["Shapes`", "Geometry`Rotations`"]`		`Shapes``	`{Shapes`, System`}`
2	`RotateShape::usage = "..."`	`Shapes`RotateShape`		
3	`Begin["`Private`"]`		`Shapes`Private``	
4	`RotateShape[shape_, ...] :=`	`Shapes`RotateShape`		
		`Shapes`Private`shape`		
5	`Module[{rotmat =`	`Shapes`Private`rotmat`		
	`RotationMatrix3D[...]},`	`Geometry`Rotations`RotationMatrix3D`		
6	`End[ ]`		`Shapes``	
7	`EndPackage[ ]`		`Global``	`{Shapes`, Geometry`Rotations`, Global`, System`}`

Table 2.2–1: Context manipulations with imported packages

Mathematica keeps track of which packages have already been read in, independently of whether they are listed on the search path `$ContextPath`. The variable `$Packages` contains a list of all contexts belonging to packages that have ever been read in. If a context specified in `Needs[]` or in an additional argument of `BeginPackage[]` is not found in `$ContextPath`, but is listed in `$Packages`, the package is not in read a second time; the context is simply put back on `$ContextPath`.

The optional arguments of `BeginPackage[]` specify those packages that are needed inside your package. Such packages are said to be *imported* in your package. Mentioning imported packages at the start of your package is also an important element of documentation. It makes all dependencies on other packages explicit. Stating exactly what your package depends on is another important principle of software design.

■ 2.2.2 Hidden Import

Any package imported through the mechanism just explained is left on the search path after your package has been read in. It is therefore also made available to the user of your package. Sometimes this is desirable or does not matter very much. But it could also hide other functions that the user of your package has defined or read in before your package. The user should not have to be concerned about other packages that are made available as a consequence of reading in yours. There is a way of making another package available to your own without leaving it on the search path after the end of your package. Instead of mentioning the package as an optional argument of `BeginPackage[]`, you can read it in with a `Needs[]` command inside your package, after the call to `BeginPackage[]`. The fact that your package imports this other package is then hidden from the user, and this method is therefore termed *hidden import*. Here is the modified code fragment of the new version of Graphics‘Shapes‘ (Listing 2.2–2), followed by the detailed analysis of the context changes (Table 2.2–2). To save some space in the table, we used the context `Shapes`` instead of `Graphics`Shapes``.

After this version of Shapes.m is read in, the context `Geometry`Rotations`` is not

```
BeginPackage["Graphics`Shapes`"]

RotateShape::usage = "..."

Begin["`Private`"]

Needs["Geometry`Rotations`"]

RotateShape[ shape_, phi_, theta_, psi_ ] :=
    Module[{rotmat = RotationMatrix3D[N[phi], N[theta], N[psi]]},
              ⋮
    ]

End[ ]

EndPackage[ ]
```

Listing 2.2–2: Hidden import in Graphics/Shapes.m

on the search path and cannot be accessed by the user. Note that inside Shapes.m the order of the contexts `Shapes`` and `Geometry`Rotations`` on the search path is reversed. This poses no problems because the current context is still `Shapes`Private``, and therefore new symbols such as `RotateShape` are still created in the correct context and not in `Geometry`Rotations``.

Even if a package is imported in this way in two different other packages, it is read only once. Although it does not remain on the search path `$ContextPath` after being read, it is remembered in the list `$Packages`, as explained in Section 2.2.1. (This did not work in earlier versions of *Mathematica*; therefore, hidden import was not widely used before.)

■ 2.2.3 Extending Other Packages

You can look at importing another package in a different way than we did in Section 2.2.1. Instead of merely using one of the functions in the imported package, you could also think of adding some more functions to the ones defined in the imported package because the

	Command	Symbols	$Context	$ContextPath
1	BeginPackage["Shapes`"]		Shapes`	{Shapes`, System`}
2	RotateShape::usage = "..."	Shapes`RotateShape		
3	Begin["`Private`"]		Shapes`Private`	
4	Needs["Geometry`Rotations`"]			{Geometry`Rotations`, Shapes`, System`}
5	RotateShape[shape_, ...] :=	Shapes`RotateShape Shapes`Private`shape		
6	Module[{rotmat = RotationMatrix3D[...]},	Shapes`Private`rotmat Geometry`Rotations`RotationMatrix3D		
7	End[ ]		Shapes`	
8	EndPackage[ ]		Global`	{Shapes`, Global`, System`}

Table 2.2–2: Context manipulations with hidden import

imported package will be available to the user of your package (remember that it remains on the context search path). Here is an example.

The package Graphics‘ParametricPlot3D‘ contains a function `ParametricPlot3D` for drawing lines in space (given in Cartesian coordinates); see Section 10.1.1. Let us now add a function for drawing lines given in *spherical* coordinates. The package SphericalCurve.m (Listing 2.2–3) effectively adds the function `SphericalCurve` to the collection of functions defined in the standard package ParametricPlot3D.m. In the version of `ParametricPlot3D[{`x`, `y`, `z`}, {`t`, `t_0`, `t_1`}]` with only one parameter t, you specify the x, y, and z coordinates of points as a function of t to generate a line in space. With `SphericalCurve[{`r`, `θ`, `φ`}, {`t`, `t_0`, `t_1`}]`, you specify the spherical coordinates r, θ, and φ in terms of a parameter t.

`SphericalCurve[]` works by converting the spherical coordinates into Cartesian coordinates and then simply calling the function `ParametricPlot3D[]`. We do not even need a local variable; therefore, no `Module[]` is necessary.

Note the form of the patterns that we use for the range of the parameter. We need the range only as a whole to pass it on to `ParametricPlot3D[]`, so we give it a name and restrict its value to a list with `ur_List`. Any additional arguments are simply passed along.

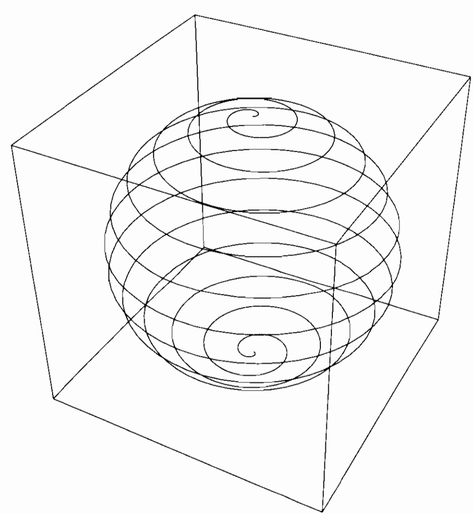
```
In[1]:= Needs[
            "ProgrammingInMathematica`SphericalCurve`"
        ]
```

The context path shows that not only ProgrammingInMathematica‘SphericalCurve‘ has been read, but also Graphics‘ParametricPlot3D‘.

```
In[2]:= $ContextPath
Out[2]= {ProgrammingInMathematica`SphericalCurve`,
  Graphics`ParametricPlot3D`,
  ProgrammingInMathematica`Options`, Global`, System`}
```

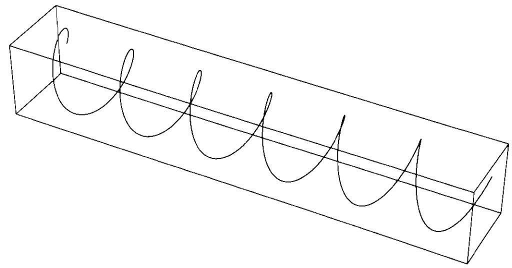
The radius is left constant (independent of `u`). The curve therefore lies on the surface of a sphere.

```
In[3]:= SphericalCurve[ {1, u/24, u},
                        {u, 0, 24Pi, Pi/24} ];
```



The functions from the imported package ParametricPlot3D.m are also available.

```
In[4]:= ParametricPlot3D[ {u/Pi, Cos[u], Sin[u]},
                          {u, 0, 12Pi, Pi/24} ];
```



```
BeginPackage["ProgrammingInMathematica`SphericalCurve`",
             "Graphics`ParametricPlot3D`"]

SphericalCurve::usage =
    "SphericalCurve[{r, theta, phi}, {u,u0,u1,(du)}, (options...)]
    plots a 3D parametric curve given in spherical coordinates."

Begin["`Private`"]

SphericalCurve[{r_, theta_, phi_}, ur_List, opts___] :=
    ParametricPlot3D[ r {Sin[theta] Cos[phi], Sin[theta] Sin[phi], Cos[theta]},
                      ur, opts ]

End[ ]

EndPackage[ ]
```

Listing 2.2–3: SphericalCurve.m: Parametric curves in spherical coordinates

■ 2.3 Protection of Symbols in a Package

A symbol is protected if it has the attribute `Protected`. No values or definitions can be added for a protected symbol. Most system symbols are protected. User-defined symbols begin their life unprotected. This section describes how to protect and unprotect symbols and how to prevent inadvertent modification of packages.

■ 2.3.1 Protecting Exported Symbols

Built-in commands are protected against accidental modification by users. Symbols can be protected and unprotected with the functions `Protect[]` and `Unprotect[]`, which are discussed in section 2.4.12 of the *Mathematica* book. Commands defined in a package can be protected in the same way and will then behave like built-in ones. All you have to do is to give the command `Protect[`*symb*$_1$`,` *symb*$_2$`, ...]` at the end of the package (between `End[]` and `EndPackage[]`), where the *symb*$_i$'s are all the symbols to be exported from the package. There is no need to protect auxiliary functions defined in the private context because they cannot be accessed by the user anyway. Here is a version of the example ComplexMap.m from Chapter 1 with added protection:

```
BeginPackage["ProgrammingInMathematica`ComplexMap`"]

CartesianMap::usage = "..."
PolarMap::usage = "..."
Lines::usage = "..."

Begin["`Private`"]

        ⋮

End[ ]

Protect[ CartesianMap, PolarMap, Lines ]

EndPackage[ ]
```

Protecting symbols in ComplexMap.m

Ordinarily, all symbols defined in a package should be protected. In this case, an easier way to protect them is to use the command

`Protect["`*Context*`` `*"]``

which protects all symbols in the given context. `Protect` accepts strings (with wildcards) as arguments and protects all symbols matching these strings. An even better construct is

`Protect[Evaluate[Context[] <> "*"]]`

(the package context is the current context between `End[]` and `EndPackage[]`; see Table 2.1–1).

A package with protected symbols should not be read in twice. The second time around the symbols would already be protected when the rules and usage messages are defined, giving you error messages. These messages are harmless; after all, the package has already been read and all definitions have been made. If you are still developing the package, however, the protection prevents new or modified definitions from being made. Therefore, you usually do not include the `Protect[]` command while a package is still under development. On the other hand, you might want to put in a statement `Clear[`*symb*$_1$, *symb*$_2$`,...]` near the beginning of the package. This makes sure that all old rules are cleared first when you modify the rules and read in the modified package into the same *Mathematica* session many times during debugging. After having found the next-to-last bug in your code, you can insert the protection commands and remove the `Clear[]` at the beginning. (The *last* bug in any piece of code is, of course, invariably found by the first user of the code and never by the programmer.) If you develop your package as a notebook you can enable or disable the `Protect[]` command easily by setting or clearing the `Evaluatable` property of the cell that contains the command (use the menu item Cell ▷ Cell Attributes to do so). If you prepare the package as an ordinary file, you can comment out the command with `(* Protect[...] *)`.

The protection of these exported symbols can still be removed by using `Unprotect[]`, just as for built-in symbols. If you really want to disable any modification, you can lock the symbols after protecting them. The attributes of a locked symbol cannot be changed at all, so this action is final (in the current *Mathematica* session). In the above example you could add the command

```
SetAttributes[{CartesianMap, PolarMap}, Locked]
```

after the call to `Protect[]`.

■ 2.3.2 Unprotecting System Symbols

A complementary case arises if a package defines additional rules for built-in functions. In this case, the affected symbols have to be unprotected at the beginning of the package and should be protected again at the end.

Here is an example package ReIm.m that defines additional rules for the built-in functions `Re[]` and `Im[]` `Im` that allow simplification of symbolic expressions involving these functions. In *Mathematica*, variables that do not have a value are treated as standing for any quantity, including complex numbers. The expression `Re[x]` therefore does not simplify to `x`. In some applications, you want to assume that variables stand for real numbers only and then you would like to simplify `Re[x]` to `x` and `Im[x]` to `0`.

To declare a variable *var* to be real, set its imaginary part to 0 with the command *var*`/: Im[`*var*`] = 0`. Rules for `Re[]` and `Im[]` defined in ReIm.m will then perform the simplifications. Listing 2.3–1 shows a few easy simplification rules for addition and multiplication.

```
BeginPackage["ProgrammingInMathematica`ReIm`"]

(* rules for simplification of Re and Im for real-value variables *)

Begin["`Private`"]

Unprotect[Re, Im]

Re[x_] := x  /; Im[x] == 0

Re[x_+y_] := Re[x] + Re[y]
Im[x_+y_] := Im[x] + Im[y]

Re[x_ y_] := Re[x] Re[y] - Im[x] Im[y]
Im[x_ y_] := Re[x] Im[y] + Im[x] Re[y]

Protect[Re, Im]

End[ ]

EndPackage[ ]
```

Listing 2.3–1: ReIm1.m: Some simplifications for real-valued variables

```
In[1]:= << ReIm1.m
```

Nothing is known about `x`. It does not simplify.

```
In[2]:= Re[x]

Out[2]= Re[x]
```

`a` is declared to be real. This declaration should be associated with `a`.

```
In[3]:= a/: Im[a] = 0;
```

It does what we expect.

```
In[4]:= Re[a]

Out[4]= a
```

This is partially simplified using the knowledge we have about `a`.

```
In[5]:= Im[a x]

Out[5]= a Im[x]
```

One feature of ReIm.m is not quite the way we want it. If the user had already unprotected `Re`, for example, then reading in our package would protect it again! This should not happen and there is a way around it. The function `Unprotect[]` returns as its value a list of all those of its arguments that it did actually unprotect. Arguments that were already unprotected are left out. Instead of protecting all the symbols again at the end, we protect only those that were returned by `Unprotect[]`. See Listing 2.3–2 for the updated package.

The return value of the `Unprotect[]` command is saved in a local variable. Its value is then used in the command `Protect[]` and only those symbols are then protected. `Protect[]` does not evaluate its arguments because it operates on the symbols themselves and not on their values. It is thus necessary to force evaluation with the command `Evaluate[]`; otherwise the symbol `protected` itself would be protected.

Here we unprotect the built-in symbol `Re`.

```
In[1]:= Unprotect[Re]

Out[1]= {Re}
```

`Re` no longer has the attribute `Protected`.

```
In[2]:= Attributes[Re]

Out[2]= {Listable, NumericFunction}
```

```
BeginPackage["ProgrammingInMathematica`ReIm`"]

Begin["`Private`"]

protected = Unprotect[Re, Im]

Re[x_] := x  /; Im[x] == 0

Re[x_+y_] := Re[x] + Re[y]
Im[x_+y_] := Im[x] + Im[y]

Re[x_ y_] := Re[x] Re[y] - Im[x] Im[y]
Im[x_ y_] := Re[x] Im[y] + Im[x] Re[y]

Protect[ Evaluate[protected] ]

End[ ]

EndPackage[ ]
```

Listing 2.3–2: ReIm.m: The final version

Im is still protected.

```
In[3]:= Attributes[Im]
Out[3]= {Listable, NumericFunction, Protected}
```

Now let us read in the package and see what happens to the protection of Re and Im.

```
In[4]:= << ProgrammingInMathematica`ReIm`
```

Both are as before.

```
In[5]:= Attributes[{Re, Im}]
Out[5]= {{Listable, NumericFunction},
  {Listable, NumericFunction, Protected}}
```

The standard package Algebra/ReIm.m defines many more such rules for simplification of `Re[]`, `Im[]`, and `Conjugate[]`.

Mathematica contains a command `ComplexExpand[]` that can be used in place of Algebra/ReIm.m in many cases. It assumes variables as real, unless declared complex. With it, the example from page 43 can be written as follows:

This simplifies `Im[a x]` assuming that `x` is complex valued and all other variables, including `a`, are real.

```
In[1]:= ComplexExpand[ Im[a x], x ]
Out[1]= a Im[x]
```

■ 2.4 Package Framework and Documentation

If you look at the examples presented so far and at other packages, you will notice that the framework of a package is the same for most of them. We developed this framework in Chapter 1 and the preceding sections of this chapter.

■ 2.4.1 A Template Package

We are now ready to collect everything together in a template or skeleton for packages. (As with real skeletons, the flesh is missing and has to be provided by the author of the package.) The cornerstones of every package are the context manipulation commands. Then come the usage messages for the functions that are to be exported, then the details about imports of other packages and the protection of symbols.

The package Skeleton.m is syntactically correct; therefore, it can be read into *Mathematica*, with `Needs["ProgrammingInMathematica`Skeleton`"]`. To avoid error messages, the three imported packages Package1.m, Package2.m, and Package3.m should be available as well. They contain only a few lines that define the contexts.

```
BeginPackage["ProgrammingInMathematica`Package1`"]
EndPackage[]
```

Package1.m: One of the imported packages

If you are writing your own package, you can take Skeleton.m (Listing 2.4–1) as a starting point. Change all the names of the functions and the package itself (including the context name in `BeginPackage[]`). You should also make sure to delete any of the features that you do not use—for example, the statements for importing other packages or for defining rules for built-in objects. As explained in Section 2.3.1, you might want to comment out the statement to protect the exported symbols while your package is still being debugged. (Under the notebook frontend, the cell containing it can be marked unevaluatable. See also Section 11.1.1 for converting this skeletal package into a notebook.)

■ 2.4.2 Headers

Documentation is an important part of programming. You can put comments in the usual form `(*` *comment* `*)` next to the code. Another important tool for documentation is the reference section of the package. This section consists of a number of standard comments, identified by keywords with colons next to them:

`(* :`*header*`:` *text*`... *).`

The standard format of these comments allows document classification tools to extract this information in a machine-readable form. The standard headers are summarized below. Some of these headers are optional and should be removed from your package if they do not apply to your particular package.

`:Title:`	title of package
`:Context:`*	the context as defined in `BeginPackage["`*context*`` ` ``"]`
`:Author:`	author's name (and affiliation)
`:Summary:`	a short abstract describing the package
`:Copyright:`*	copyright notice: © *year* by *name*
`:Package Version:`	package version number in the form $n.n$
`:Mathematica Version:`	the lowest *Mathematica* version required
`:History:`*	one-line description of earlier versions and change log
`:Keywords:`	words most useful for document retrieval, separated by commas
`:Sources:`*	references to the literature consulted
`:Warnings:`*	incompatibilities, global effects
`:Limitations:`*	known problems, special cases not handled
`:Discussion:`*	more information for expert users, description of algorithms
`:Requirements:`*	other packages or files needed (including imported packages)
`:Examples:`*	sample input that demonstrates the features of the package

Standard headers suggested for packages (* optional header)

If you plan to submit your package to *MathSource*, these headers are important to properly classify and retrieve your package among the gigabytes of information present.

Further documentation tools are offered by the frontend; see Section 11.1.4.

```
(* :Title: Skeleton.m -- a package template *)

(* :Context: ProgrammingInMathematica`Skeleton` *)

(* :Author: Roman E. Maeder *)

(* :Summary:
   The skeleton package is a syntactically correct framework for package
   development.
 *)

(* :Copyright: © <year> by <name or institution> *)

(* :Package Version: 2.0 *)

(* :Mathematica Version: 3.0 *)

(* :History:
   2.0 for Programming in Mathematica, 3rd ed.
   1.1 for Programming in Mathematica, 2nd ed.
   1.0 for Programming in Mathematica, 1st ed.
*)

(* :Keywords: template, skeleton, package *)

(* :Sources:
   Roman E. Maeder. Programming in Mathematica, 3rd ed. Addison-Wesley, 1996.
*)

(* :Warnings:
   <description of global effects, incompatibilities>
*)

(* :Limitations:
   <special cases not handled, known problems>
*)

(* :Discussion:
   <description of algorithm, information for experts>
*)

(* :Requirements:
   ProgrammingInMathematica/Package1.m
   ProgrammingInMathematica/Package2.m
   ProgrammingInMathematica/Package3.m
*)

(* :Examples:
   <sample input that demonstrates the features of this package>
*)

(* set up the package context, including public imports *)

BeginPackage["ProgrammingInMathematica`Skeleton`",
  "ProgrammingInMathematica`Package1`", "ProgrammingInMathematica`Package2`"]

(* usage messages for the exported functions and the context itself *)

Skeleton::usage = "Skeleton.m is a package that does nothing."

Function1::usage = "Function1[n] does nothing."
Function2::usage = "Function2[n, (m:17)] does even more nothing."

(* error messages for the exported objects *)

Skeleton::badarg = "You twit, you called `1` with argument `2`!"
```

```
Begin["`Private`"]    (* begin the private context (implementation part) *)
Needs["ProgrammingInMathematica`Package3`"]    (* read in any hidden imports *)
(* unprotect any system functions for which definitions will be made *)
protected = Unprotect[ Sin, Cos ]
(* definition of auxiliary functions and local (static) variables *)
Aux[f_] := Do[something]
staticvar = 0
(* definition of the exported functions *)
Function1[n_] := n
Function2[n_, m_:17] := n m /; n < 5 || Message[Skeleton::badarg, Function2, n]
(* definitions for system functions *)
Sin/: Sin[x_]^2 := 1 - Cos[x]^2
Protect[ Evaluate[protected] ]     (* restore protection of system symbols *)
End[ ]         (* end the private context *)
Protect[ Function1, Function2 ]    (* protect exported symbols *)
EndPackage[ ]  (* end the package context *)
```

Listing 2.4–1: Skeleton.m: A template for package development

■ 2.5 Loading Packages

Packages are additions to the standard *Mathematica* functionality that can be loaded as needed. This section discusses the correspondence between context names (meaningful to *Mathematica*) and file names (meaningful to the operating system) that is employed to find packages and treats a particular problem that arises if functions in a package are referenced before the package has been loaded. Packages can also be set up to be loaded on demand, whenever a function from them is used. This setup allows for a seamless integration of the new functionality into *Mathematica*.

■ 2.5.1 Context Names and Package Names

When a context name is specified in a `Needs["`*Context*`` ` ``"]` command or as one of the optional arguments of `BeginPackage[]`, *Mathematica* has to derive the name of a file to read. By convention, the name of the file is of the form *Context*.m where *Context* is the context name without the final `` ` ``. If the context name is composed of several parts separated by context marks, then these context marks are translated into appropriate path separators for your file system (/ under UNIX, for example). The command ``Needs["Geometry`Rotations`"]`` would try to read in the package Geometry/Rotations.m. This is the usual case, because all standard packages are put into subdirectories of the Package directory.

The translation between context names and file names is performed by the function `ContextToFileName["`*Context*`` ` ``"]` that is set up correctly on each different machine. Its basic function is to replace context marks by pathname separators and to append the standard extension for packages. The pathname separator is taken from the global variable `$PathnameSeparator`.

Under MS-DOS and systems derived from it, `ContextToFileName[]` also truncates context names to the infamous eight-character limit of file names. As a result, contexts that do not differ in their first eight characters are mapped to the same file name.

To further allow machine-independent programming, the `Get[]` command also accepts a context name instead of a file name. You can therefore read in the package Geometry/Rotations.m, for example, using ``<<Geometry`Rotations` ``. Under UNIX, this context name would then be translated back to Geometry/Rotations.m. See also Subsection 2.11.5 of the *Mathematica* book.

We turn on tracing for the function oiContextToFileName[].

```
In[1]:= On[ContextToFileName]
```

When we specify a context name as argument of `Get[]` we see that it calls `ContextToFileName[]` to obtain a valid file name for the computer currently in use. The second call is from the imported package.

```
In[2]:= << Graphics`Shapes`

ContextToFileName::trace:
   ContextToFileName[Graphics`Shapes`] —>
    Graphics/Shapes.m.

ContextToFileName::trace:
   ContextToFileName[Geometry`Rotations`] —>
    Geometry/Rotations.m.
```

`Needs[]` has an optional second argument that allows you to specify a different file name. This, however, is not possible in the `BeginPackage[]` command. One way around it is to precede `BeginPackage[]` by a call to `Needs[]`. If the package context ``Rotations` `` is in the file TestRotations.m instead of Rotations.m, the following piece of code shows a possible way to import this package into Shapes.m in the same way as was done in Section 2.2.1:

```
Needs["Rotations`", "TestRotations.m"]

BeginPackage["Graphics`Shapes`", "Rotations`"]
        ⋮
EndPackage[ ]
```

Using a file name different from the context name

Files derived from contexts are searched relative to the file search path, `$Path`. It contains a list of directories possibly containing packages or directories of packages. An example is the directory containing the standard packages. It is usually named AddOns/StandardPackages and can be found in the directory where *Mathematica* was installed.

The search path contains a number of standard directories and possibly others that were added in the user's init.m file. The entry `"."` stands for the current directory.

```
In[1]:= $Path

Out[1]= {., /home/bellatrix/maeder,

  /usr/local/Mathematica/AddOns/StandardPackages,

  /usr/local/Mathematica/AddOns/StandardPackages/StartUp,

  /usr/local/Mathematica/AddOns/Applications,

  /usr/local/Mathematica/AddOns/ExtraPackages,

  /usr/local/Mathematica/SystemFiles/Graphics/Packages}
```

Let us visit the standard packages directory.

```
In[2]:= SetDirectory[ $Path[[3]] ]

Out[2]= /usr/local/Mathematica/AddOns/StandardPackages
```

It contains these subdirectories.

```
In[3]:= FileNames[]

Out[3]= {Algebra, Calculus, DiscreteMath, Geometry,

  Graphics, LinearAlgebra, Miscellaneous, NumberTheory,

  NumericalMath, Statistics, Utilities}
```

We restore our previous current directory.

```
In[4]:= ResetDirectory[ ];
```

As we have seen, ``Needs["Geometry`Rotations`"]`` first derives the file name Geometry/Rotations.m. Then, it searches all directories in `$Path` for the presence of such a file. The first one found is read in.

■ 2.5.2 Shadowing of Symbols

Putting an exported symbol in a separate context has one problem if that symbol already exists in the global context. Here is a *Mathematica* session that illustrates the problem.

We try to use a function, but forgot to read in the package first. *Mathematica* does not know about this function, so it returns our input. But it has created the symbol `CylindricalPlot3D` in the global context.

```
In[1]:= CylindricalPlot3D[ 1.5 Sqrt[1 + r^2],
                           {r, 0, 2}, {phi, 0, 2Pi} ]

                                       2
Out[1]= CylindricalPlot3D[1.5 Sqrt[1 + r ], {r, 0, 2},

  {phi, 0, 2 Pi}]
```

We want to correct the mistake and read in the package that defines `CylindricalPlot3D[]`. We are warned that the new symbol `CylindricalPlot3D` will be shadowed by the already existing one.

```
In[2]:= << Graphics`ParametricPlot3D`

CylindricalPlot3D::shdw:
   Symbol CylindricalPlot3D appears in multiple contexts
    {Graphics`ParametricPlot3D`, Global`}; definitions in
     context Graphics`ParametricPlot3D`
     may shadow or be shadowed by other definitions.
```

Now let's try again. It still does not work!

```
In[3]:= CylindricalPlot3D[ 1.5 Sqrt[1 + r^2],
                           {r, 0, 2}, {phi, 0, 2Pi} ]

                                       2
Out[3]= CylindricalPlot3D[1.5 Sqrt[1 + r ], {r, 0, 2},

  {phi, 0, 2 Pi}]
```

The symbol `CylindricalPlot3D` in the global context is found first and hides the one that was defined in ParametricPlot3D.m.

```
In[4]:= Context[ CylindricalPlot3D ]

Out[4]= Global`
```

This command asks for all symbols with name `CylindricalPlot3D` in all contexts. There are indeed two of them. (The first one is not printed as ``Global`CylindricalPlot3D`` because it can be accessed without typing its context.)

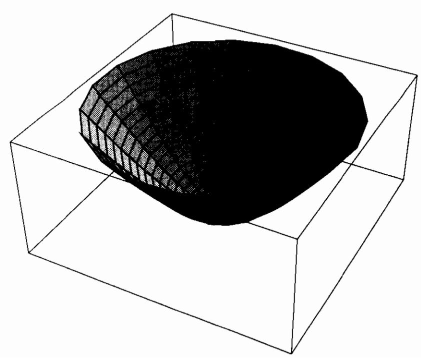
```
In[5]:= ?*`CylindricalPlot3D

CylindricalPlot3D
Graphics`ParametricPlot3D`CylindricalPlot3D
```

You could use the function by always referring to it as ``Graphics`ParametricPlot3D`Cylindrical-Plot3D``, but that would be awkward. A drastic action is to remove the offending global symbol.

```
In[6]:= Remove[ CylindricalPlot3D ]
```

Now it finds the correct one because the symbol in the global context no longer exists, and we finally get our hyperboloid.

```
In[7]:= CylindricalPlot3D[ 1.5 Sqrt[1 + r^2],
                           {r, 0, 2}, {phi, 0, 2Pi} ];
```



In cases where we anticipate that this problem may occur frequently, there is a nice trick to avoid it. Simply mention `Global`` as one of the contexts to be imported into your package! No file will be read because this context already exists. The effect is that the global context will stay on the context path during the time that the package is read in. Any definitions for a symbol that exists already in the global context will be attached to that symbol, not to a new one (see Section 2.1.2).

■ 2.5.3 Autoloading Packages

The unpleasant situation described in Section 2.5.2 can be avoided altogether by automatically loading a package when a function from it is used the first time. The command

`DeclarePackage["`*Context*`'", {"`sym_1`", "`sym_2`", ..., "`sym_n`"}]`

causes `Needs["`*Context*`'"]` to be called whenever one of the symbols sym_i is entered the first time. The list of symbols should comprise all symbols exported from the given package. Note that the symbols must be given as strings. Let us look at our example Geometry/Rotations.m. Its package context is `"Geometry'Rotations'"` and it exports the symbols `RotationMatrix2D`, `Rotate2D`, `RotationMatrix3D`, and `Rotate3D`.

This command can be put into your init.m to cause autoloading on demand of Geometry/Rotations.m.

```
In[1]:= DeclarePackage["Geometry'Rotations'",
            {"RotationMatrix2D", "Rotate2D",
             "RotationMatrix3D", "Rotate3D"} ];
```

It works by creating the required symbols and giving them the attribute `Stub`. The package is not yet loaded, however.

```
In[2]:= ?RotationMatrix2D

RotationMatrix2D[theta] gives the matrix for rotation by
   angle theta in two dimensions.
```

The first time one of the stub symbols is used, the package is loaded and no error occurs.

```
In[3]:= RotationMatrix2D[ 20.0 Degree ]

Out[3]= {{0.939693, 0.34202}, {-0.34202, 0.939693}}
```

It would be too cumbersome to extract the contexts and symbols from a package by hand, to write the `DeclarePackage[]` statements. *Mathematica* can help us with this task.

Here is the context for which we want to create a `DeclarePackage[]` statement.

```
In[4]:= context = "Geometry'Rotations'"

Out[4]= Geometry'Rotations'
```

We load the package, if necessary.

```
In[5]:= Needs[context]
```

Here is a list of all symbols exported, that is, all symbols in the package context. Although this is not visible here, the entries in the list are strings. The pattern `**` matches all names, including names that begin with a dollar sign.

```
In[6]:= names = Names[context <> "**"]

Out[6]= {Rotate2D, Rotate3D, RotationMatrix2D,
  RotationMatrix3D}
```

We open the Autoload.m file to which we shall write the `DeclarePackage[]` statement.

```
In[7]:= file = OpenWrite[ "Autoload.m",
                          FormatType -> InputForm ]

Out[7]= OutputStream[Autoload.m, 5]
```

The command is written out, unevaluated, of course. The construct `With[{var = var}, body]` inserts the current value of the variable *var* everywhere in *body*, even in parts that are not evaluated (see Section 5.6.1).

```
In[8]:= With[{context = context, names = names},
          Write[ file,
            Unevaluated[DeclarePackage[context, names]]
          ] ]
```

We close the file.

```
In[9]:= Close[file]

Out[9]= Autoload.m
```

Here is the contents of our sample autoloading file:

```
DeclarePackage["Geometry`Rotations`",
  {"Rotate2D", "Rotate3D", "RotationMatrix2D", "RotationMatrix3D"}]
```

Autoload.m

Note that autoloading does not work for packages that redefine system symbols (such as the package Relm.m described in Section 2.3.2). No new symbols exist that could be used in the `DeclarePackage` command. To load such a package, put the command `Needs["`*package*`` ` ``"]` into your init.m file.

■ 2.5.4 Master Packages

The package MakeMaster.m contains the command

`MakeMaster[` *file*, `{`*context*$_1$, ..., *context*$_n$`}` `]`

that performs the steps explained in the previous subsection and writes autoloading commands for all the given contexts to *file*. The code is shown in Listing 2.5–1. If you use a small number of packages (your own or standard ones) frequently, putting such autoloading commands into your init.m is the recommended way to autoload these packages.

There is another approach for autoloading *all* packages from a directory of packages. You may have noticed that every directory in the standard packages directory of *Mathematica* contains a file named Master.m and an init.m file in a subdirectory named Kernel. Both of these files contain `DeclarePackage[]` statements for all packages in its directory. (Master.m is provided for backward compatibility with Version 2.2.) To use the autoloading package, simply put `Needs["`*directory*`` ` ``"]` into your initialization file. *Mathematica* searches the given directory first for Kernel/init.m, then for init.m, and loads the first file found. These master packages do not export any symbols at all, but they nevertheless set up a dummy package context that helps keep track of which ones are already loaded (see Section 2.2.2 for an explanation of how `Needs[]` figures out whether a package needs to be loaded).

A second rule in our `MakeMaster[]` generates such master packages for a whole directory of packages. It is used like this:

`MakeMaster[` *masterfile*, *directory* `]`.

The default master file name is init.m. Directories are interpreted relative to `$Path` because only in this way will the packages be found when `Needs[]` is used to autoload them later. The current directory can be specified as `""` or `"."`.

The method to create such a master package requires the following steps (see Listing 2.5–1).

- The desired directory is found on the search path with `FileNames[`*directory*`, $Path]`. The result is a list of all matching (absolute) directory names. If there are none, we return with an error. If there are more than one, we print a warning and use the first one.

  If the desired directory is the current directory, this step is skipped.

- All packages (files ending in .m) in the desired directory are found. For this, we temporarily set the current directory to the place where we found *directory* (using `SetDirectory[]` and `ResetDirectory[]`). If there is already a file named init.m or Master.m we do not include it in the list of all files.

- The file names are converted to contexts with `filenameToContext[`*filename*`]`, which is the inverse of the standard function `ContextToFileName["`*context*`` ` ``"]`. Note how it uses `StringReplace[]` to convert path separators and the .m suffix to context marks.

- Now we have a list of contexts and can use the other rule for `MakeMaster[]` to complete the task. The option `PackageContext` is given to set the correct value of the dummy context in the "package" init.m.

Note that the various constants, such as context mark, package name suffix, or path separators, are assigned to local variables. This idea makes the code easier to maintain. Whenever possible, the values are obtained in a machine-independent way. Even the context mark (which is unlikely to change) is obtained as the last character of an actual context (the current context, found in `$Context`) and not as the constant string ``"`"``.

Of course, the directory with the packages for this book, ProgrammingInMathematica, contains an init.m file, made with the tools from this section. It does not contain declarations for intermediate versions of packages, only for the final ones. The file is shown in Listing 2.5–2.

If a context given in a `Get[]` command corresponds to a directory, the init.m file in this directory is read. Here, all packages for our book are preloaded.

```
In[1]:= << ProgrammingInMathematica`
```

The context path now contains all packages from the book. The packages themselves have not been read yet, however.

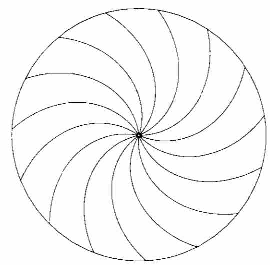
```
In[2]:= Short[ $ContextPath, 4 ]
Out[2]//Short=
 {ProgrammingInMathematica`VectorCalculus`,
  ProgrammingInMathematica`Until`,
  ProgrammingInMathematica`TrigSimplification`,
  ProgrammingInMathematica`TrigDefine`, <<22>>, Global`,
  System`}
```

When DeclarePackage[] is executed from within init.m, the current value of $Input is temporarily assigned to all stub symbols created. This value is used to locate the package defining the symbol if no package is found on $Path.

```
In[3]:= ?PolarMap
ProgrammingInMathematica`ComplexMap`PolarMap
Attributes[PolarMap] = {Stub}
PolarMap = "ProgrammingInMathematica.m"
```

As soon as a function is used, the corresponding package is first read in (ComplexMap.m in this case).

```
In[4]:= PolarMap[Function[z, Abs[z] Exp[I(Arg[z]+Abs[z])]],
                {0.01, 1}, Lines -> {2,16}, Axes -> None];
```



```
BeginPackage["ProgrammingInMathematica`MakeMaster`"]

MakeMaster::usage = "MakeMaster[file, {contexts..}] writes DeclarePackage
    commands for all given contexts to file. MakeMaster[file, directory]
    writes a master file for all packages in directory. Directory is
    interpreted relative to $Path. The current directory can be given as \".\"."
PackageContext::usage = "PackageContext -> context is an option of
    MakeMaster that gives the context to use for the master package."

Begin["`Private`"]

(* possibly OS-dependent constants *)
$packageExtension = ".m"
$contextMark = StringTake[$Context, -1]
$master = "init"
$masterName = $master <> $packageExtension
$masterFormat = InputForm    (* format for writing master file *)
$oldMasterName = "Master" <> $packageExtension    (* V2.2 files *)

Options[MakeMaster] = { PackageContext :> $master <> $contextMark }

MakeMaster[ filename_String:$masterName, contexts_List, opts___?OptionQ ] :=
    Module[{loaded, file},
        Needs /@ contexts; (* load them *)
        loaded = Union @@
            Function[cont, Select[$ContextPath, StringMatchQ[#, cont]&]] /@
            contexts;
        file = OpenWrite[filename, FormatType -> $masterFormat];
        If[ file === $Failed, Return[file] ];
        (* preamble *)
        With[{cont = PackageContext /. {opts} /. Options[MakeMaster]},
            Write[file, OutputForm["(* Created by MakeMaster *)"]];
            Write[file, Unevaluated[BeginPackage[cont]]];
            Write[file, Unevaluated[EndPackage[]]];
            Write[file, OutputForm[""]];
        ];
```

```
        makeMaster[ file, #]& /@ loaded;
        Write[file, Null]; Close[file]
    ]

MakeMaster[ filename_String:$masterName, directory_String ] :=
    Module[{contexts, cont},
        contexts = contextsForDirectory[directory];
        If[contexts === $Failed, Return[contexts] ];
        cont = If[ directory == "" || directory == ".", $masterName,
                   directory <> $PathnameSeparator <> $masterName ];
        MakeMaster[ filename, contexts,
            PackageContext -> filenameToContext[cont] ]
    ]

filenameToContext[filename_String] :=
    Module[{cont},
        cont = StringReplace[filename, {$PathnameSeparator -> $contextMark,
                                        $packageExtension -> $contextMark}];
        If[ StringTake[cont, -1] != $contextMark, cont = cont <> $contextMark ];
        cont
    ]

makeMaster[ file_, context_String ] :=
    With[{names = Names[context <> "**"]},
        If[names =!= {}, Write[file, Unevaluated[DeclarePackage[context, names]]]]
    ]

(* current directory is special *)
contextsForDirectory["" | "."] :=
    Module[{files},
        files = Complement[FileNames["*" <> $packageExtension], {$masterName}];
        filenameToContext /@ files
    ]

contextsForDirectory[directory_String] :=
    Module[{absdir, files},
        absdir = FileNames[directory, $Path];
        If[ Length[absdir] == 0, Message[MakeMaster::nodir, directory];
                                 Return[$Failed] ];
        If[ Length[absdir] > 1,  Message[MakeMaster::sdir, directory] ];
        absdir = First[absdir];
        SetDirectory[ ParentDirectory[absdir] ];
        files = Complement[FileNames["*" <> $packageExtension, directory],
                          {directory <> $PathnameSeparator <> $masterName,
                           directory <> $PathnameSeparator <> $oldMasterName}];
        ResetDirectory[];
        filenameToContext /@ files
    ]

MakeMaster::nodir = "No directory matching `` found on $Path."
MakeMaster::sdir = "Warning: more than one directory matching `` found on $Path."

End[]

Protect[MakeMaster]

EndPackage[]
```

Listing 2.5–1: MakeMaster.m: Creating master packages

```
(* Created by MakeMaster *)
BeginPackage["ProgrammingInMathematica`init`"]
EndPackage[]

DeclarePackage["ProgrammingInMathematica`AffineMaps`",
  {"AffineMap", "AverageContraction", "map", "rotation", "scale",
   "translation", "$CirclePoints"}]
DeclarePackage["ProgrammingInMathematica`AlgExp`", {"AlgExpQ"}]
DeclarePackage["ProgrammingInMathematica`Atoms`", {"Explode", "Intern"}]
DeclarePackage["ProgrammingInMathematica`ChaosGame`",
  {"ChaosGame", "Coloring"}]
DeclarePackage["ProgrammingInMathematica`Collatz`",
  {"Collatz", "FindMaxima", "StoppingTime"}]
DeclarePackage["ProgrammingInMathematica`ComplexMap`",
  {"CartesianMap", "Lines", "PolarMap", "$Lines"}]
DeclarePackage["ProgrammingInMathematica`ContinuedFraction`",
  {"CF", "CFValue"}]
DeclarePackage["ProgrammingInMathematica`FoldRight`",
  {"FoldLeft", "FoldLeftList", "FoldRight", "FoldRightList"}]
DeclarePackage["ProgrammingInMathematica`IFS`",
  {"ifs", "IFS", "Probabilities"}]
DeclarePackage["ProgrammingInMathematica`MakeFunctions`",
  {"LinearFunction", "MakeRule", "MakeRuleConditional", "StepFunction"}]
DeclarePackage["ProgrammingInMathematica`MakeMaster`",
  {"MakeMaster", "PackageContext"}]
DeclarePackage["ProgrammingInMathematica`Newton`",
  {"NewtonFixedPoint", "NewtonZero"}]
DeclarePackage["ProgrammingInMathematica`NotebookLog`", {"NotebookLog"}]
DeclarePackage["ProgrammingInMathematica`Options`",
  {"SetAllOptions", "SymbolsWithOptions"}]
DeclarePackage["ProgrammingInMathematica`RandomWalk`", {"RandomWalk"}]
DeclarePackage["ProgrammingInMathematica`RungeKutta`", {"RKSolve"}]
DeclarePackage["ProgrammingInMathematica`SessionLog`",
  {"CloseLog", "OpenLog"}]
DeclarePackage["ProgrammingInMathematica`SphericalCurve`", {"SphericalCurve"}]
DeclarePackage["ProgrammingInMathematica`Struve`", {"StruveH"}]
DeclarePackage["ProgrammingInMathematica`SwinnertonDyer`",
  {"SwinnertonDyerP"}]
DeclarePackage["ProgrammingInMathematica`Tensors`", {"li", "Tensor", "ui"}]
DeclarePackage["ProgrammingInMathematica`TrigDefine`", {"TrigDefine"}]
DeclarePackage["ProgrammingInMathematica`TrigSimplification`",
  {"TrigArgument", "TrigLinear"}]
DeclarePackage["ProgrammingInMathematica`Until`", {"Until"}]
DeclarePackage["ProgrammingInMathematica`VectorCalculus`",
  {"Div", "Grad", "JacobianMatrix", "Laplacian"}]
Null
```

Listing 2.5–2: init.m: The master file for the book packages

2.6 Large Projects

A larger software projects requires a higher level of organization than a single package can provide. We look at ways of splitting an application into several packages and discuss installation issues for applications.

2.6.1 Package Directories

A larger application, consisting of several packages, deserves its own directory (or folder, as a directory is called in some operating systems). Putting all files belonging to a larger project into their own directory makes it easier to maintain and distribute the files. The context defined for the packages should be adapted to this organization of the files. Contexts can be hierarchical, just like directories. The package package.m in the directory myproject, for example, should declare the package context `myproject`package``. That is, the call to `BeginPackage[]` (see Section 2.1) looks like this

```
BeginPackage["myproject`package`"].
```

The package can be loaded into a *Mathematica* session with

```
Needs["myproject`package`"].
```

Note that the directory *containing* the myproject directory must be on the file search path `$Path`. On systems supporting the notion of a home directory, your home directory is usually on this search path, so you can simply create the subdirectory myproject in your home directory. There is also a standard place for such package directories in the *Mathematica* distribution directory. It is the directory AddOns/Applications. If you copy your directory myproject into this directory, *Mathematica* will find the files as described. If you prepare an application for distribution, you should instruct your users to install the package directory in a standard place, such as the home directory or the AddOns/Applications subdirectory of the *Mathematica* installation. The former is more appropriate for multi-user environments, the latter for typical single-user PCs. The packages described in this book, can be installed in a directory called ProgrammingInMathematica, either in your home directory, or in AddOns/Applications or in AddOns/ExtraPackages. (see installation instructions on page xvi).

If a package contains the command `BeginPackage["`*name₁*`` ` ``*name₂*`` ` ``"]` the directory *containing* the *name*$_1$ directory should be on the search path, not *name*$_1$ itself.

Mathematica uses the same organization for its standard packages. The package directory consists of subdirectories such as Algebra or Graphics. The package Shapes.m

in the Graphics directory, for example, defines the context `Graphics`Shapes`` in its `BeginPackage[]` command.

You can use `FileNames[`*name*`,` *directorylist*`]` to verify that a particular file or directory *name* is indeed on the search path `$Path`. Here, the Graphics subdirectory of the standard packages is found.

```
In[1]:= FileNames[ "Graphics", $Path ]
Out[1]= {/usr/local/Mathematica/AddOns/StandardPackages/G\
   raphics}
```

Because the directory containing Graphics is on the search path, we can load the Shapes package in this way.

```
In[2]:= Needs[ "Graphics`Shapes`" ]
```

To find out which package is actually loaded, you can turn on tracing for `ContextToFile-Name`, as explained in Section 2.5.1.

Once you have developed a directory full of packages you should consider generating a master package for it to allow the packages to be autoloaded (see Section 2.5.3). The command to do so is simply

```
MakeMaster["myproject"] .
```

This command will generate a file init.m in the current directory; you should move it into the myproject directory. This master file can be read simply with `<<myproject``. All packages from the directory are then preloaded.

If the myproject directory is put in AddOns/Autoload instead of AddOns/Applications, the master file is read automatically when *Mathematica* starts up. By asking your users to install your directory in this place, you can avoid many problems caused by forgotten initializations.

When the stub symbols for all packages in an application are created by reading in init.m, the location of the master package is remembered. This feature allows the packages defining the symbols to be found even if they are not on the search path `$Path` later on, when the symbols are used for the first time. This may happen if the application is installed in the autoload directory instead of the applications directory (the former is usually not on the search path), or if the init.m file is loaded directly from the CD-ROM on which the application is distributed. (See also Section 2.5.4.)

■ 2.6.2 Common Packages

The standard packages have been organized into directories according to subject matter. In some of these directories (Algebra, for example) the individual packages are completely independent; in other directories, such as Statistics, they are interrelated. You will find a subdirectory named Common inside the Statistics directory. The packages it contains are not meant to be used by themselves; rather, they are auxiliary packages needed by the packages in the Statistics directory. As explained in Section 2.2, there are essentially two ways to import a package into another package: public import and hidden import.

■ 2.6.2.1 Public Import of Common Packages

If the common package provides functionality of interest to the end users of the main package, it should be imported publicly, by mentioning it in the `BeginPackage[]` command. Here is the outline of a package that publicly imports an auxiliary package:

```
BeginPackage["myproject`package`", "myproject`Common`auxpackage`"]
⋮
Begin["`Private`"]
⋮
End[]
EndPackage[]
```

Note that the context for the auxiliary package has an additional component, because it is found in a subdirectory of the myproject directory. The auxiliary package looks like this:

```
BeginPackage["myproject`Common`auxpackage`"]
⋮
EndPackage[]
```

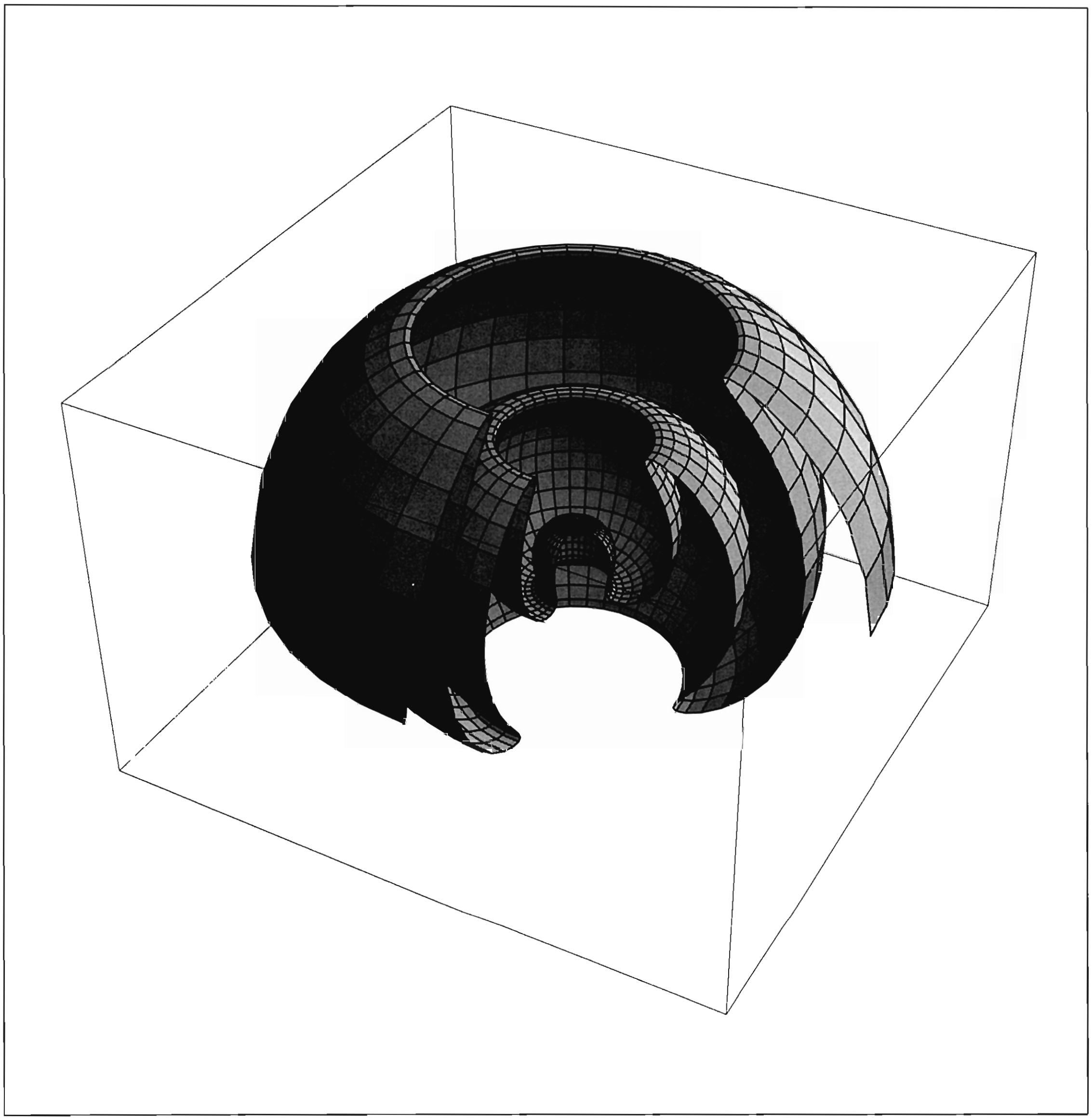
■ 2.6.2.2 Hidden Import of Common Packages

If the auxiliary package does not provide any functionality of interest to the end user, it should be imported in the implementation part. Here is the outline of a package that uses hidden import for an auxiliary package:

```
BeginPackage["myproject`package`"]
⋮
Begin["`Private`"]
Needs["myproject`Common`auxpackage`"]
⋮
End[]
EndPackage[]
```

Chapter 3

Defaults and Options



In a programming environment that is mainly used interactively, the design of the user interface of functions that you write is of great importance. *Mathematica* provides two mechanisms for designing an easy-to-use interface.

The first one allows you to leave out parameters that you do not want to specify in simpler applications. Many built-in functions have default values for some of their arguments. These defaults are chosen to give the most basic form of the function in question. Novice users will not even have to know about these arguments; the function does the "right thing."

The second feature—options—is used if a function has many parameters that should be user-settable. In such a case, defaults would be too confusing. Options have a name and so their use is rather intuitive, while defaults rely on separate documentation about their placements and values. You can look at the default values of options and reset them globally.

Section 1 presents the basics about defaults for arguments. It also makes clear their limitations, which motivate us to find ways to overcome them. This advanced treatment of defaults is treated next.

In Section 2 we discuss how to define and use options for your own functions in the same way that they are used in built-in functions, for example, the graphics functions. We present also a useful utility for picking out options in an option list when we want to pass those options on to functions called within our function.

Finally, Section 3 presents tools for changing defaults for options of several commands at once.

About the illustration overleaf:

A cut-out view of a rotationally symmetric parametric surface. The command to generate this picture is

```
ParametricPlot3D[
   {r Cos[Cos[r]] Cos[psi], r Cos[Cos[r]] Sin[psi], r Sin[Cos[r]]},
   {r, 0.001, 9Pi/2}, {psi, 0, 3Pi/2}, PlotPoints -> {72, 30}]
```

The equation of the generating curve in the x–z plane is $\varphi = \cos r$.

■ 3.1 Default Values

Defaults are values of parameters of functions that are used when the corresponding parameter is left out in a function call. Used sparingly, they can help to avoid repetitive typing. Default values are always constants, and special coding techniques are needed to implement more complicated cases, where the defaults depend on other parameters of the function. We shall look at these issues after reviewing ordinary defaults.

■ 3.1.1 The Syntax of Defaults

To give a default value to a "blank" in a pattern, you simply use x_:*def*, where *def* is the value that *Mathematica* should assume for the blank, named x, if it is left out from an expression. You can experiment with *Mathematica* to see the internal form of an expression with `FullForm[`*expr*`]`.

This is an optional pattern named `x` with default `1`.

```
In[1]:= FullForm[ x_:1 ]
Out[1]//FullForm= Optional[Pattern[x, Blank[]], 1]
```

Defaults can be given only to simple blanks (and blank sequences). Here are the two forms that are possible.

x_:*def*	an expression named x with default value *def*
x_h:*def*	an expression with head h, named x and default value *def*

The two forms of defaults for simple pattern variables

There is, however, another use of the colon in patterns; it is of the form t:*pat* and is used to give a name to a complicated pattern. Section 2.3 of the *Mathematica* book discusses patterns. (The full syntax of patterns is given in Subsection A.5.1 of that book.)

Here is a pattern, given the name `t`.

```
In[2]:= FullForm[ t:_ ]
Out[2]//FullForm= Pattern[t, Blank[]]
```

In the simplest case of giving a name to a blank, the colon is optional. This input gives the same expression.

```
In[3]:= FullForm[ t_ ]
Out[3]//FullForm= Pattern[t, Blank[]]
```

Here is a pattern matched by a list of to elements. The whole matching expression is named `t`.

```
In[4]:= FullForm[ t:{a_, b_} ]
Out[4]//FullForm=
 Pattern[t, List[Pattern[a, Blank[]],
   Pattern[b, Blank[]]]]
```

The parser cannot understand this expression that is supposed to denote a pattern matched by a list of two elements whose defaults should be 0 and 1, respectively.

```
In[5]:= FullForm[ {a_, b_}:{0, 1} ]
Syntax::sntxf:
   "FullForm[ {a_, b_}" cannot be followed by ":{0, 1} ]".
```

The ambiguity can be resolved by naming the whole pattern.

```
In[5]:= FullForm[ t:{a_, b_}:{0, 1} ]
Out[5]//FullForm=
 Optional[Pattern[t, List[Pattern[a, Blank[]],
    Pattern[b, Blank[]]]], List[0, 1]]
```

t:*pat*	a pattern *pat*, named *t*
t:*pat*:*def*	a pattern *pat*, named *t* with default value *def*

Defaults and names of patterns

3.1.2 Possible Values of Defaults

The default value you specify is evaluated at the time the pattern is defined (in the left side of a rule normally). It cannot contain names of other patterns. This is a consequence of the way the right side of a rule is used. The pattern names are replaced by their values in parallel.

We try to define a function with a default value for its second argument. That value should be one less than the first argument. The function does nothing but return that value so we can check whether it works.

```
In[6]:= f[ n_, m_:(n-1) ] := m
```

However, this does not work.

```
In[7]:= f[2]
Out[7]= -1 + n
```

It would be possible to make it work with this obscure construction. Using global variables in this way is, however, considered bad programming style.

```
In[1]:= f[n_, m_:x] := Block[ {x = n-1}, m ]
```

The default value of `m` is `x`, but inside the block `x` has a value and this value is then used.

```
In[2]:= f[2]
Out[2]= 1
```

This simple kind of default is useful for constant default values, but not for more complicated cases. For defaults that depend on other parameters of a function, we need a different setup, which is the topic of the next subsection.

3.1.3 Computed Defaults: Using a Token

Recall from Section 3.1.1 that a simple default of the form *pat*:*def* is a constant, evaluated when the definition is made.

We have seen the preferred way of specifying computed defaults in Section 1.5.1. The default value in the pattern is a special symbol whose presence is then checked inside the body of the rule in an If[] statement.

```
f[a_, b_, cv_:Automatic] :=
    Module[ {c = cv},
        If[ c === Automatic, c = ... ];  (* compute default *)
        ⋮
    ]
```

A template for computed defaults

The use of an extra local variable (c in this case) is necessary because names of patterns (cv in this case) cannot be used like local variables in the body of the rule (see Section 5.1.1). The symbol Automatic serves as a token to detect the use of the default value. It is important that it does not have a value. Automatic is a built-in symbol and therefore protected, so everything is fine.

■ 3.1.4 Giving Several Rules

Another way of defining defaults was discussed in Section 1.4. We can define a separate rule for each case we want to consider. One rule is the main one containing all the code that implements our function. It requires all arguments to be present. Other rules have an argument list leaving some of the arguments of the main rule out. Their body is usually short. They compute the appropriate value for the left out arguments and call themselves again. Here is the general layout:

```
f[a_, b_, c_] :=
    Module[ {...},
            ⋮          (* main code goes here *)
          ]

f[a_, b_] :=
    Module[ {c},
        c = ... ;    (* compute value for c *)
        f[a, b, c];  (* call main routine   *)
    ]
```

Separate rules for default arguments

As an example, let us add a new rule to our package ComplexMap.m from Chapter 1. When using the function PolarMap[], the range for the angular variable will frequently be {0, 2Pi}, going once around the circle. We want to use this range by default. To do so, we add a rule for PolarMap[] that leaves out the second range specifier completely. The new and final version of the package is ComplexMap.m (excerpted in Listing 3.1–1, shown in full in Listing 1.6–2).

```
PolarMap::usage = "PolarMap[f, {r0:0, r1, (dr)}, {phi0, phi1, (dphi)}] plots the
    image of the polar coordinate lines under the function f. The default for
    the phi range is {0, 2Pi}. The default values of dr and dphi are chosen
    so that the number of lines is equal to the value of the option Lines."

⋮

PolarMap[ func_, {r0_:0, r1_, dr_:Automatic}, {p0_, p1_, dp_:Automatic},
          opts___?OptionQ ] :=
    Module[ {r, p}, Picture[ PolarMap, func[r Exp[I p]],
                             {r, r0, r1, dr}, {p, p0, p1, dp}, opts ]
    ] /; NumericQ[r0] && NumericQ[r1] && NumericQ[p0] && NumericQ[p1] &&
     (NumericQ[dr] || dr === Automatic) && (NumericQ[dp] || dp === Automatic)

PolarMap[ func_, rr_List, opts___?OptionQ ] := PolarMap[ func, rr, {0, 2Pi}, opts ]
⋮
```

Listing 3.1–1: ComplexMap.m (excerpt): Separate rules for default values

Observe that we can use a simpler form to match the radial range `rr` in the second rule because all we want to do with it is to pass it along.

We read in the final version of the complex map package.

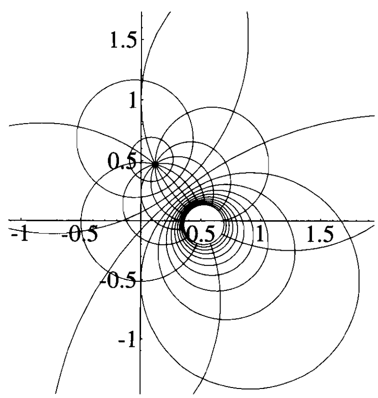
```
In[1]:= << ProgrammingInMathematica`ComplexMap`
```

This is the minimum information we have to specify, using all the defaults, including the start value 0 for the radius. The function

$$f(z) = \frac{z - i}{2z - 2 - i/2}$$

used in this example is called a *Möbius transform.*

```
In[2]:= PolarMap[Function[z, (z - I)/(2z - 2 - I/2)], {4}];
```



When you give several rules for the same function, you should think about the correct ordering of them. With complicated argument lists, *Mathematica* cannot always find out which of the rules is more special than the other and it might fail to reorder them accordingly. To verify that the ordering is correct, you should convince yourself that the argument list of a particular rule is not already matched by an earlier one. If it were, then these two rules should be exchanged.

■ 3.2 Options for Your Functions

Options are named parameters in function calls that can be given in any order, or left out if a default value should be used. Built-in functions make heavy use of options, especially graphics functions. This section tells you how to define options for your own functions. (An introduction to this subject appeared in Section 1.4.)

■ 3.2.1 Commands for Defining Options

Let us first review how you use options for built-in functions. The names and the default values of options are predefined. The command `Options[`*f*`]` returns a list of all options defined for the symbol *f* together with their default values. The result is returned as a list of rules, and this form will be useful when using options for our own functions. The command `SetOptions[]` is used to change these defaults.

`Options[`*f*`]`	list all options defined for *f*
`SetOptions[`*f*`,` opt_1 `->` val_1`, ...]`	change default values of options for *f*

The two functions for dealing with options

To define options for our own function *g*, we simply make an assignment for the expression `Options[`*g*`]`. This assignment is automatically stored with the symbol *g* rather than with `Options`. Once this is done we can use the function `SetOptions[]` in the same way as with built-in functions. This is briefly mentioned in Subsection 2.3.10 of the *Mathematica* book. A first example of defining options for a function was also given in Section 1.4.1.

This defines the names and initial default values of the options for `g`. Usually you should use `=` for this assignment rather than `:=`.

```
In[1]:= Options[g] = {Opt1 -> val1, Opt2 -> val2}
Out[1]= {Opt1 -> val1, Opt2 -> val2}
```

This works just as it does for built-in functions.

```
In[2]:= Options[g]
Out[2]= {Opt1 -> val1, Opt2 -> val2}
```

Mathematica tells us what it knows about `g`. The assignment from input line 1 has indeed been stored under the symbol `g`.

```
In[3]:= ?g
Global`g
Options[g] = {Opt1 -> val1, Opt2 -> val2}
```

This resets the default for the second option.

```
In[4]:= SetOptions[g, Opt2 -> newval]
Out[4]= {Opt1 -> val1, Opt2 -> newval}
```

Full error checking is done. Only previously defined options can be changed.

```
In[5]:= SetOptions[g, Opt3 -> val3]
SetOptions::optnf: Opt3 is not a known option for g.
Out[5]= SetOptions[g, Opt3 -> val3]
```

■ 3.2.2 Using Options in a Function

We now have a framework for dealing with options from the user's point of view. Now we look at the programmer's side. Inside our function, we need to look at the values that have been specified for each option, either the default value (if the option is not given on the parameter list) or the value given by an argument of the form *opt* `->` *value*.

`Options[`*cmd*`] = {`*opt* `->` *val*`, ...}`	define the options and their defaults for a command
cmd`[` *args*`..., opts___?OptionQ]`	a template for the argument list of a command taking options
var `=` *opt* `/. {opts} /. Options[`*cmd*`]`	obtain the user-specified or default value to use in the body of a command

Code for defining and reading options in a command

Because options are given as rules, we use the standard substitution operation *expr* `/.` *rule* for extracting the value of an option. On the argument list, the options are specified by the pattern `opt___?OptionQ` to match any sequence of options, including the empty sequence. Listing 3.2–1 shows the outline of code necessary to find the values of options for a function.

```
BeginPackage["ProgrammingInMathematica`OptionUse`"]

g::usage = "g[n, options...] serves as an example for using options."

Opt1::usage = "Opt1 is an option of g[]."
Opt2::usage = "Opt2 is another option of g[]."

Options[g] = {Opt1 -> val1, Opt2 -> val2}

Begin["`Private`"]

g[ n_, opts___?OptionQ ] :=
    Module[ {opt1, opt2},
        opt1 = Opt1 /. {opts} /. Options[g];
        opt2 = Opt2 /. {opts} /. Options[g];
        {n, opt1, opt2}
    ]

End[]

EndPackage[]
```

Listing 3.2–1: OptionUse.m: The use of options in a package

Let us understand how the local variables `opt1` and `opt2` get their values. If `g` is called in the form `g[5]` without specifying any options, then the list `{opts}` is the empty list and the expression `Opt1 /. {opts}` evaluates to `Opt1` because no rules were given on the right side of the substitution operator. The expression `Options[g]`, however, is always a list of rules, one for each option, as we have seen. So the result of

`Opt1 /. {opts} /. Options[g]` is the current default value for the option `Opt1` which is then assigned to the local variable `opt1`. Note that substitutions are grouped to the left.

If, however, g is called in the form `g[5, Opt1 -> newval]` then the list `{opts}` is equal to `{Opt1 -> newval}` and the substitution `Opt1 /. {opts}` evaluates to `newval`. The following substitution `newval /. Options[g]` does nothing, because none of the rules matches. So the result of `Opt1 /. {opts} /. Options[g]` is the value for the option `Opt1` specified in the argument list which is then assigned to the local variable `opt1`.

Options should be documented just like functions. We have added such documentation to the example and also included the package framework. The names of the options are symbols in the package context. Our example function does nothing exciting. It simply returns the values of its required argument and the values of the options that are actually used inside the function. It is useful for playing around with options to understand how they work.

```
In[1]:= << ProgrammingInMathematica`OptionUse`
```

Both options take on their default value.

```
In[2]:= g[5]
Out[2]= {5, val1, val2}
```

We can specify a different value for one of the options.

```
In[3]:= g[5, Opt1 -> 17]
Out[3]= {5, 17, val2}
```

We can also set a new default, as seen in Section 3.2.1.

```
In[4]:= SetOptions[g, Opt2 -> newdef]
Out[4]= {Opt1 -> val1, Opt2 -> newdef}
```

This new value is now used instead of the old one.

```
In[5]:= g[5]
Out[5]= {5, val1, newdef}
```

If the same option is given twice, the first value encountered on the argument list is used.

```
In[6]:= g[5, Opt2 -> value1, Opt2 -> value2]
Out[6]= {5, val1, value1}
```

Any rule for a nonexisting option is simply ignored.

```
In[7]:= g[5, Opt3 -> value3]
Out[7]= {5, val1, newdef}
```

■ 3.2.3 Inheriting Options

The package Graphics/Shapes.m defines a command `Polyhedron[`*name*`]` that returns a `Graphics3D` object representing a polyhedron. As such, it can possibly accept any valid option of `Graphics3D` and insert it into the resulting data structure (options of graphics objects are stored in a list as the second element of the objects). The code to do so is simple:

```
Polyhedron[ name_Symbol, opts___?OptionQ ] :=
    Graphics3D[ rendering[name], Flatten[{opts}] ]
```

The function `rendering[]` is assumed to compute the list of graphics primitives that describe the polyhedron. How the polyhedra are computed is described in Section 4.6.

This idea has the limitation that no separate defaults for the options of `Polyhedron` can be set. You cannot, for example, set the default of `PlotRange` to `All` for polyhedra,

without disturbing the default for other `Graphics3D` objects; because `PlotRange` is not an option of `Polyhedron`, you cannot use

```
SetOptions[ Polyhedron, PlotRange -> All ].
```

The solution is to define all graphics options as options of `Polyhedra`, too. Care must be taken to put all their values into the `Graphics3D` object; otherwise, the defaults defined for `Graphics3D` would be used, instead of the defaults defined for `Polyhedron`. The new code fragment looks like this:

```
Options[Polyhedron] = Options[Graphics3D]

SetOptions[ Polyhedron, PlotRange -> All ] (* modified default *)

Polyhedron[ name_Symbol, opts___?OptionQ ] :=
    Graphics3D[ rendering[name], Flatten[{opts, Options[Polyhedron]}] ]
```

Part of Graphics/Polyhedra.m

As you can see, we simply assign the options of `Graphics3D` to those of `Polyhedron`. If we want to define a different default for some of them, we can use `SetOptions` in the package, as we did for `PlotRange`. The code of `Polyhedron[]` always inserts *all* options into the second element of the `Graphics3D` structure. Note that any options given on the command line come before this list of all options. Otherwise they would never take effect, because the first occurrence of an option is used.

■ 3.2.4 Filtering Options

In Section 1.4 we saw how the functions `CartesianMap[]` and `PolarMap[]` handle the option `Lines` and pass any other options on to the embedded graphics commands `ParametricPlot3D[]` and `Show[]`. Because these graphics commands allow different sets of options (and, especially, do not handle the option `Lines`), we needed a way of filtering options. This subsection explains how the auxiliary package Utilities/FilterOptions.m works.

We need a function that takes as arguments the name of a command and a sequence of options and returns only those of the given options that are valid for this command, discarding all others.

`FilterOptions[`*cmd*`, `*options...*`]`	return only those options that are valid for the command *cmd*
`FilterOptions[{`*opts...*`}, `*options...*`]`	return only those options whose names are among *opts*

A function for filtering option sequences

```
BeginPackage["Utilities`FilterOptions`"]

FilterOptions::usage = "FilterOptions[symbol, options..] returns a sequence
    of those options that are valid options for symbol.
    FilterOptions[{opts..}, options..] filters out options with names opts."

Begin["`Private`"]

FilterOptions[ command_Symbol, options___ ] :=
    FilterOptions[ First /@ Options[command], options ]

FilterOptions[ opts_List, options___ ] :=
    Sequence @@ Select[ Flatten[{options}], MemberQ[opts, First[#]]& ]

End[ ]

Protect[ FilterOptions ]

EndPackage[ ]
```

Listing 3.2–2: Utilities/FilterOptions.m

The functional programming style allows us to write this function in a very compact way; see Listing 3.2–2.

When called in the form `FilterOptions[`*symbols*`, `*options*`...]`, the names of the options for the command *symbols* are determined, and the function is called again. This time, the second definition matches and performs the actual computation. To show you how it works, we use a simple debugging technique. We assign sample parameters to the names of the patterns in the argument list for `FilterOptions[]` and then step through the body of the function, unwinding the nested function calls along the way. The symbol `Sequence` is perhaps new to you. Sequences are indeed peculiar objects, and a discussion of them is deferred to Section 5.3.5. Suffice it to say that the expression substituted for a pattern of the form *name*`___` has as its head the symbol `Sequence`. Let us now understand what goes on when we evaluate the expression

```
FilterOptions[Graphics, Axes->None, PlotPoints->33, Frame->True].
```

We assign the value of the first parameter to its name.

```
In[1]:= command = Graphics
Out[1]= Graphics
```

We do likewise for the sequence of options. Here we need the symbol `Sequence`.

```
In[2]:= options = Sequence[ Axes -> None, PlotPoints -> 33,
                            Frame -> True ]
Out[2]= Sequence[Axes -> None, PlotPoints -> 33,
  Frame -> True]
```

Here we unwind the computation of the first argument for the second call of `FilterOptions`. First we get the list of options for our command. Graphics functions tend to have many options, and our list gets rather long.

```
In[3]:= Options[command] // Short
Out[3]//Short=
                          1
 {AspectRatio -> -----------, Axes -> False,
                 GoldenRatio
  AxesLabel -> None, <<21>>, TextStyle :> $TextStyle}
```

Then, we extract the first element of each of the options. This gives a list of the *names* of the options. This list becomes the value of the pattern variable opts in the second call to FilterOptions.

```
In[4]:= Short[ opts = First /@ %, 3 ]
Out[4]//Short=
 {AspectRatio, Axes, AxesLabel, AxesOrigin, AxesStyle,
  Background, ColorOutput, <<14>>, DefaultFont,
  DisplayFunction, FormatType, TextStyle}
```

This is the result of mapping the predicate that occurs as the second argument of Select[] to our sequence of options. The predicate checks whether the names of these options occur in the list opts. The second one of them is not a valid option for Graphics[]. The other two are.

```
In[5]:= MemberQ[opts, First[#]]& /@ Flatten[{options}]
Out[5]= {True, False, True}
```

This performs the actual selection according to the values of the predicate from the line above. Only the first and third of the options are selected.

```
In[6]:= Select[ Flatten[{options}],
                MemberQ[opts, First[#]]& ]
Out[6]= {Axes -> None, Frame -> True}
```

This replaces the head List by Sequence, making the result suitable for splicing into the options part of another function.

```
In[7]:= Sequence @@ %
Out[7]= Sequence[Axes -> None, Frame -> True]
```

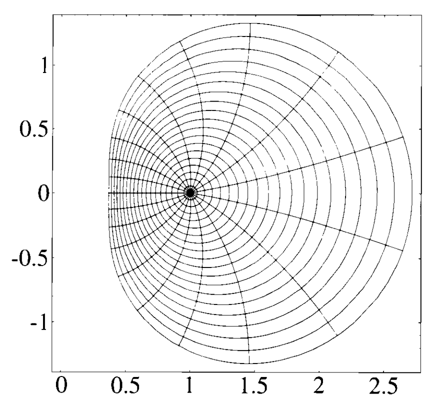
The reason we use Flatten[{options}] instead of simply {options} is that option specifications may be (nested) lists of options. Although we never make use of this form of options in our book, this utility should support all admissible forms of options.

It was easy to incorporate FilterOptions[] into the package ComplexMap.m from Chapter 1. We used hidden import (see Section 2.2.2) because FilterOptions[] is of no use by itself. The graphic function for which the options of CartesianMap[] and PolarMap[] are to be filtered is Graphics. Refer to Listing 1.6–2 for the code.

```
In[8]:= << ProgrammingInMathematica`ComplexMap`
```

The options Axes and Frame are passed to Show[]. The setting for Lines takes effect immediately and is not passed on. PlotPoints is passed to ParametricPlot3D[].

```
In[9]:= PolarMap[ Exp, {1}, {-Pi, Pi},
                  Axes -> None, Frame -> True,
                  Lines -> 20, PlotPoints -> 30 ];
```



If a command *Cmd* has the attribute `HoldAll` or `HoldRest`, splicing of options in the form

Cmd`[`*args*`, FilterOptions[`*Cmd*`, `*opts*`]]`

does not work. Use

Cmd`[`*args*`, Evaluate[FilterOptions[`*Cmd*`, `*opts*`]]]`

in this case to force evaluation and splicing of the option sequence before *Cmd* takes over.

■ 3.3 Setting Options of Several Commands

The same option can appear in several related or unrelated commands. Graphics commands, for example, share many options. There are no built-in tools for finding all commands that understand a given option or for setting the default value for all occurrences of an option. Let us develop such tools.

■ 3.3.1 Global Variables as Defaults

One way to allow the default to be changed in all commands that use a certain option is to use an indirect default. An example is the option `DisplayFunction` used by all graphics commands. Its default is not a particular display function, but the value of the global variable `$DisplayFunction`.

The value of the option `DisplayFunction` is the global variable `$DisplayFunction`.

```
In[1]:= Options[ Graphics, DisplayFunction ]
Out[1]= {DisplayFunction :> $DisplayFunction}
```

The value of this global variable is the real default. Its actual value is system dependent.

```
In[2]:= $DisplayFunction
Out[2]= Display[$Display, #1] &
```

An assignment to `$DisplayFunction` causes *every* graphic function to use the new value.

```
In[3]:= $DisplayFunction = Identity;
```

Note that the default is defined using a delayed rule (`:>` instead of `->`). This is important to prevent the current value (when the option defaults are defined at program startup) from being literally inserted into the option settings of the graphic commands.

Let us demonstrate this technique using our package ComplexMap.m from Chapter 1. The two commands `CartesianMap[]` and `PolarMap[]` use the same option, `Lines`, for essentially the same purpose. We shall define a global variable `$Lines` that is used

```
⋮
Lines::usage = "Lines -> {lx, ly} is an option of CartesianMap and PolarMap
    that gives the number of lines to draw."
$Lines::usage = "$Lines is the default of the option Lines. The value should be
    either a positive integer or a list of two positive integers."
⋮
$Lines = 15; (* global default *)

Options[CartesianMap] = Options[PolarMap] = { Lines :> $Lines }
⋮
```

Listing 3.3–1: An excerpt from ComplexMap.m

as a common default. The relevant code, excerpted from ComplexMap.m, is shown in Listing 3.3–1.

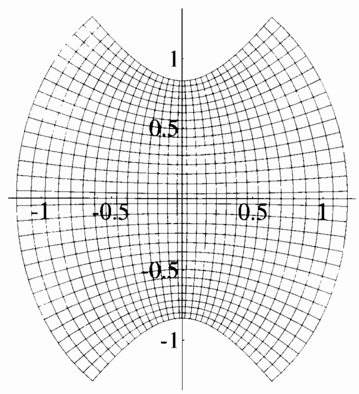
The new global variable `$Lines` should be documented and it must be assigned an initial default value. The assignments to the option lists must be changed to the form `Lines :> $Lines`.

One assignment is all it takes to change the defaults for both `CartesianMap[]` and `PolarMap[]`.

```
In[1]:= $Lines = 30;
```

The new default is used.

```
In[2]:= CartesianMap[ Sinh, {-1, 1}, {-1, 1} ];
```



Global variables, such as `$Lines`, must not be protected at the end of the package. The protection would make it impossible to change their values, defeating the purpose for which these variables are used in the first place.

■ 3.3.2 All Commands That Know a Given Option

Our second tool for changing option defaults of several related commands at once is a command `SetAllOptions[]` that finds all symbols that understand a given option and then changes the default of all of them.

`SetAllOptions[` opt_1 `->` val_1`, ... ]`	change the defaults for all commands that know about the options opt_1, opt_2, ...

The command `SetAllOptions`

The idea is that we first find all symbols whose options include opt_1, opt_2, ..., then call `SetOptions[`*sym*`,` opt_1 `->` val_1`,` opt_2 `->` val_2`, ...]` on all those symbols. Let us demonstrate the idea in an interactive session that will change the setting of `PlotPoints` for all commands in the ``System` `` context that know the option `PlotPoints`.

Assume that we want to set the default of `PlotPoints` to 50 for all commands that know about the option `PlotPoints`.

```
In[1]:= arg = (PlotPoints -> 50);
```

First, we extract the name of the desired option, `PlotPoints` in our example.

```
In[2]:= opt = First[ arg ]
Out[2]= PlotPoints
```

Here, we get the names (as strings) of all symbols in the system context.

```
In[3]:= Names[ "System`" <> "*" ] // Short
Out[3]//Short=
 {Abort, AbortProtect, Above, Abs, AbsoluteDashing,
  <<1551>>, $UserName, $Version, $VersionNumber}
```

We convert the strings to symbols, but we must be careful not to evaluate them (because some symbols may have values, for example `$RecursionLimit`). `ToExpression[`*string*`, InputForm, Hold]` converts a string into an expression and immediately wraps it in `Hold[]`, before any evaluation is done.

```
In[4]:= ToExpression[#, InputForm, Hold]& /@ %  // Short
Out[4]//Short=
 {Hold[Abort], Hold[AbortProtect], Hold[Above], <<1554>>,
  Hold[$Version], Hold[$VersionNumber]}
```

Eventually, we want to get rid of the inner `Hold[]` around the symbols. Therefore, we first replace the list by a structure that is not evaluated, such as `Hold[]`.

```
In[5]:= Hold @@ %  // Short
Out[5]//Short=
 Hold[Hold[Abort], Hold[AbortProtect], Hold[Above],
  <<1554>>, Hold[$Version], Hold[$VersionNumber]]
```

Even though the contents of `Hold[]` are not evaluated, we can still use a replacement rule to remove the inner `Hold[]`.

```
In[6]:= (allSymbols = % /. Hold[sym_] :> sym)  // Short
Out[6]//Short=
 Hold[Abort, AbortProtect, Above, Abs, AbsoluteDashing,
  <<1551>>, <<8>>e, $Version, $VersionNumber]
```

We select all those symbols that do have options, that is, whose list of options is not empty. Note the attribute `HoldFirst` given to the pure function to prevent the evaluation of its argument and the use of `Unevaluated` to prevent the evaluation of the argument of `Options[]`. (Pure functions are explained in Section 5.2.)

```
In[7]:= Select[ allSymbols,
            Function[sym,
                     Length[Options[Unevaluated[sym]]] > 0,
                     {HoldFirst}]
        ] // Short
Out[7]//Short=
 Hold[AccountingForm, AlgebraicRules, Apart,
  ApartSquareFree, Apply, <<153>>, Variables, Zeta]
```

The protection from evaluation is no longer needed because symbols with options are commands that do not have values.

```
In[8]:= List @@ %  // Short
Out[8]//Short=
 {AccountingForm, AlgebraicRules, Apart, ApartSquareFree,
  Apply, <<152>>, Union, Variables, Zeta}
```

Here, we select all those symbols that know about the option `opt`. To do so, we get the list of options and apply `First` to these options to get their names. Then, we select those for which `opt` is a member of this list of option names.

```
In[9]:= syms = Select[ %,
            Function[sym,
                     MemberQ[First /@ Options[sym], opt]]
        ]
Out[9]= {ContourPlot, DensityPlot, ParametricPlot,
  ParametricPlot3D, Plot, Plot3D}
```

Finally, we call the function `SetOptions` for all symbols filtered out above. `Scan[]` is like `Map[]`, but it does not return a list of results (which we do not need here).

```
In[10]:= Scan[ Function[sym, SetOptions[sym, arg]], syms ]
```

To check the result, we look at the default value of `PlotPoints` for the command `DensityPlot`. It has indeed been changed as desired.

```
In[11]:= Options[ DensityPlot, PlotPoints ]
Out[11]= {PlotPoints -> 50}
```

The package Options.m, shown in Listing 3.3–2, implements a refinement of the ideas we just developed interactively. The computation has been broken up into a number of small auxiliary functions, and these improvements have been added:

- The function `SymbolsInContext["`*Context*`` ` ``"]` finds all symbols in the given context.
- The list of *all* symbols is the union of the list of symbols in all interesting contexts. These are all contexts on the search path `$ContextPath`. The context search path lists all contexts in which symbols are found. We do not want to search for symbols in other contexts.
- There are two definitions for the auxiliary function `symbolsWithOptions[]`. The first one handles a single option in the form `symbolsWithOptions[`*symbols*`, `*opt*`]` and filters out from the list of symbols *symbols* all those that know about the option *opt*.

  The second definition `symbolsWithOptions[`*symbols*`, `*options*`]` accepts a list of options and filters out all symbols that know about *all* options in the list *options* by repeatedly filtering out one option from the list. `Fold[`f`, `v_0`, {`v_1`, ..., `v_n`}]` iteratively applies the binary function f to the previous result and the v_i in turn, starting with initial value v_0 (see Section 4.4.4). The function f in our case is `symbolsWithOptions`.
- The two rules for `SymbolsWithOptions` implement the two different ways this function can be called: either with a list of option names as argument in the form `SymbolsWithOption[{`opt_1`, ..., `opt_n`}]`, or by giving the option names as separate arguments as `SymbolsWithOption[`opt_1`, ..., `opt_n`]`. As long as no ambiguity is introduced, a little bit of user-friendliness cannot hurt.
- `SetAllOptions[]` restricts its argument to a sequence of options with the predicate `OptionQ`. Because `OptionQ` accepts also *lists* of options, we use `Flatten[]` to remove any inner list braces that may be present.
- The variable `allSymbols` is defined with a delayed definition (`:=`). Therefore, the list of all symbols is *recomputed* every time it is needed (inside `allSymbolsWithOptions` and eventually inside `SetAllOptions[]`). This recomputation is necessary because the list of symbols may change, especially each time a new package is read in.

`SetAllOptions[]` is particularly useful for customizing your *Mathematica* environment in an init.m file.

This command can be used to change the output of all graphics commands to gray level, for example, for preparing a manuscript such as this one (see also page xv).

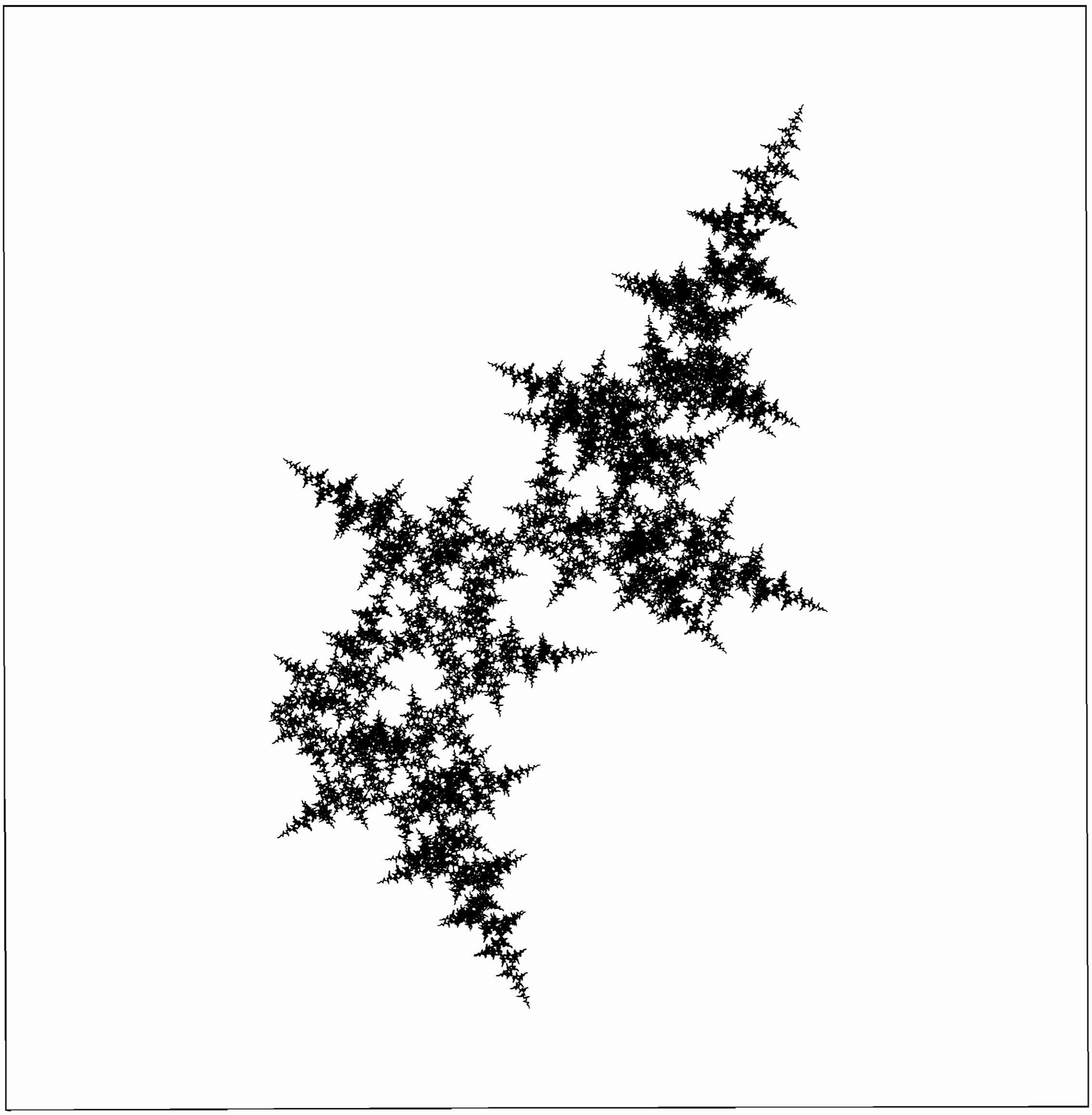
```
In[1]:= SetAllOptions[ ColorOutput -> GrayLevel ]
Out[1]= {ContourGraphics, ContourPlot, DensityGraphics,
  DensityPlot, Graphics, Graphics3D, GraphicsArray,
  ListContourPlot, ListDensityPlot, ListPlot, ListPlot3D,
  ParametricPlot, ParametricPlot3D, Plot, Plot3D,
  SurfaceGraphics}
```

```
BeginPackage["ProgrammingInMathematica`Options`"]

SymbolsWithOptions::usage = "SymbolsWithOptions[opt1, opt2, ...] gives a list of
    all symbols that know about the options named opt1, opt2, ..."
SetAllOptions::usage = "SetAllOptions[opt1 -> val1, opt2 -> val2, ...] sets the
    given options for all commands that know about all of these options."

Begin["`Private`"]

SymbolsInContext[context_String] :=
        Hold @@ (ToExpression[#, InputForm, Hold]&) /@ Names[context <> "*"] /.
        Hold[sym_] :> sym

allSymbols := Join @@ SymbolsInContext /@ $ContextPath

allSymbolsWithOptions :=
    List @@ Select[ allSymbols,
                    Function[sym, Length[Options[Unevaluated[sym]]] > 0,
                                  {HoldFirst}] ]

symbolsWithOptions[ symbols_List, opts_List ] :=
        Fold[ symbolsWithOptions, symbols, opts ]

symbolsWithOptions[ symbols_List, opt_Symbol ] :=
    Select[ symbols, Function[sym, MemberQ[First /@ Options[sym], opt]] ]

SymbolsWithOptions[ opts_List ] :=
    symbolsWithOptions[ allSymbolsWithOptions, opts ]

SymbolsWithOptions[ opts__ ] := SymbolsWithOptions[ {opts} ]

SetAllOptions[ args___?OptionQ ] :=
    With[{syms = SymbolsWithOptions[First /@ Flatten[{args}]]},
         Scan[ Function[sym, SetOptions[sym, args]], syms];
         syms
    ]

End[]

Protect[ SymbolsWithOptions, SetAllOptions ]

EndPackage[]
```

Listing 3.3–2: Options.m: Auxiliary functions for manipulating options

Chapter 4

Functional and Procedural Programming



Mathematica allows you to program in a variety of styles. The most commonly identified programming styles are procedural programming, functional programming, declarative programming, and object-oriented programming. If you have done your programming so far in one programming language exclusively, I could probably tell you which one you are used to by looking at your first *Mathematica* programs.

Rather than just continue in the style you are used to, you should learn how to use *Mathematica*'s commands in the best possible way, choosing the style that is best suited to the problem to be solved. Here, we put the emphasis on procedural and functional programming. In Chapter 6 we shall look in detail into the unique style called *mathematical programming* that *Mathematica* offers.

The first section deals with localization and information hiding at the level of individual procedures. Declaring auxiliary variables *local* to a procedure prevents conflicts with values of the arguments passed to it.

Section 2 deals with the basic forms of iteration, available in almost any programming language. The next section then introduces some iteration constructs that are unique. They correspond very closely to the way we think about mathematics and should be used whenever possible.

Often we can avoid programming a loop altogether by applying functions to lists or other expressions. Section 4 and the following section about mapping of functions over expressions go to the heart of *Mathematica*'s programming language. Understanding this material will allow you to write concise programs in the style that the authors of *Mathematica* think is best to use.

In Section 6 we apply some of these ideas to an example. We develop functions to display and manipulate three-dimensional regular polyhedra.

Finally, Section 7 looks at useful functions that deal with nested lists or matrices, including transpositions, rotations, and generalized inner and outer products.

About the illustration overleaf:

A sequence of unit vectors whose direction is $ci^2 \bmod 2\pi$, for $i = 1, 2, \ldots$ and $c = \frac{\pi}{2}\left(1+\sqrt{5}\right)$. Values of c that are irrational multiples of π give nonrepeating figures, such as this one. The picture is due to an idea by J. Waldvogel.

```
c = Pi(1 + Sqrt[5])/2.0;
x = Range[50000];
ListPlot[{Re[#], Im[#]}& /@ FoldList[Plus, 0, Exp[I c x^2]],
      PlotJoined -> True, AspectRatio -> Automatic, Axes -> None]
```

4.1 Procedures and Local Variables

Contexts and packages organize the larger units of a program. They were described in Chapter 2. We now turn to the smaller units, the procedures and functions that make up a larger program. The principles of information hiding, abstraction, and encapsulation of internal workings apply on this level, too.

4.1.1 Procedures in *Mathematica*

Strictly speaking, there are no procedures, functions, or subroutines in *Mathematica*. Any definition of the form `f[`*args*`] :=` *body* is a rewrite rule. Whenever the evaluator sees an expression that matches the left side, it is replaced by the right side with the values of the pattern variables substituted. This corresponds closely to a procedure call of a traditional language, a similarity that is intended. The benefit of this setup is that a parameter list of a procedure is not restricted to the usual `proc[`arg_1`, `arg_2`, ..., `arg_n`]`, but can be any pattern. The procedure `CartesianMap[]` from Chapter 1, for example, uses the form (or *calling sequence*)

```
CartesianMap[ f_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic}, opts___ ] .
```

It declares optional arguments, a variable number of extra arguments (options), and combines the two ranges into lists, thus adding clarity to the argument structure. Depending on the form of such a rewrite rule, we usually call it a procedure, function, or transformation rule.

```
SplitLine[vl_] :=
    Module[{vll, pos, linelist = {}, low, high},
        vll = If[NumberQ[#], #, Indeterminate]& /@ vl;
        pos = Flatten[ Position[vll, Indeterminate] ];
        pos = Union[ pos, {0, Length[vll]+1} ];
        Do[ low = pos[[i]]+1;
            high = pos[[i+1]]-1;
            If[ low < high, AppendTo[linelist, Take[vll, {low, high}]] ],
           {i, 1, Length[pos]-1}];
        linelist
    ]
```

A typical procedure

```
RandomPoly[x_, n_] := Sum[ Random[Integer, {-10, 10}] x^i, {i, 0, n} ]
```

A typical function

```
log[a_ b_] := log[a] + log[b]
```

A typical transformation rule

■ 4.1.2 Local Variables

Local variables are variables that exist only within a procedure definition (or rewrite rule in *Mathematica*). Using local variables to hold intermediate results within the body of a procedure is an important program design principle. We have already seen that the use of *global* variables can lead to interaction between the procedure and the rest of the program (see Section 1.2). The part of the program in which a declared variable is visible and can be used is called the *scope* of the variable. A *block* is a part of a program that declares local variables whose scope is that part of the program. A language that supports blocks of scope is called a block-structured language. Examples are ALGOL 60, **C**, and PASCAL. FORTRAN also allows local variables, but scopes cannot be nested.

The construct `Module[{`*variables*`}, `*body*`]` declares the variables in the list *variables* as local. Their scope is the expression *body*, usually a compound statement of the form *stmt*$_1$`;` *stmt*$_2$`;` ...`;` *stmt*$_n$`.` `Block[]` also introduces local variables. It is used for special purposes. The differences are discussed in Section 5.6. An example where both are used within the same function is the command `ShowTime[]` in Section 8.1.2.

A `Module` is most often used as the body of a procedure, such as `SplitLine` above. The template for such procedures is

```
f[...] :=
    Module[{...},
        stmt1;
        ⋮
    ]
```

The similarity with procedure definitions in languages such as **C** is intended.

In most block-structured languages, formal parameters of a procedure are treated as initialized local variables, often called "call-by-value." When the procedure is called they are initialized with the values of the actual arguments passed to the procedure. Some languages also allow "call-by-reference" parameters. Any occurrence of such a parameter stands for the actual argument. Any assignment to it modifies the actual argument. *Mathematica* has no special mechanism for procedure parameters, but uses pattern matching. Superficially, the names given to patterns play a role similar to parameters, but they are not variables. The value of the actual argument is simply *substituted* for every occurrence of the pattern variable in the body of the procedure. Their behavior, therefore, is similar to that of call-by-reference parameters. For a detailed discussion of formal parameters, see Section 5.1.1.

4.2 Loops

Loops and iterations are fundamental to any programming language. There are two basic kinds of iterations: repeating a statement a fixed number of times and iterating statements as long as a some condition is satisfied. This section introduces the commands `Do[]`, `While[]`, and `For[]` that are available in one way or another in all procedural programming languages. As we shall see later, *Mathematica* offers alternatives that are better suited for many applications.

4.2.1 Iteration

The most basic form of a loop is `Do[`*statement*`,` *iterator*`]`. It allows you to perform a statement over and over again with the iterator variable taking on successive values. The `Do[]` statement itself returns no value.

This prints the squares of the first five positive integers. The output is a side-effect of the `Print[]` statement. No value is returned; there is no `Out[]` line.

```
In[1]:= Do[ Print[i^2], {i, 1, 5} ]

1
4
9
16
25
```

Sometimes you do not need the loop variable at all. The piece of code shown in Listing 4.2–1 computes the n^{th} Fibonacci number f_n. Each Fibonacci number is defined as the sum of the previous two. The first and second one are defined to be 1. The sequence begins with 1, 1, 2, 3, 5, 8, 13, ...

```
fibonacci[n_Integer?Positive] :=
    Module[{fn1=1, fn2=0},
        Do[ {fn1, fn2} = {fn1 + fn2, fn1}, {n-1} ];
        fn1
    ]
```

Listing 4.2–1: Fibonacci1.m: Iterative computation of Fibonacci numbers

The local variables `fn1` and `fn2` are initialized to the first two Fibonacci numbers. In the loop we repeatedly replace `fn1` with the sum of the previous two numbers that are stored in `fn1` and `fn2` at any time. `fn2` is then set to the old value of `fn1`. The parallel assignment of both `fn1` and `fn2` makes this quite easy without the use of an additional local variable. Before the loop, `fn1` is initialized to $f_1 = 1$. Therefore, the loop has to be repeated $n - 1$ times to compute f_n.

This gives the Fibonacci number f_{100}. Loops of this kind are quite fast.

```
In[1]:= fibonacci[100]
Out[1]= 354224848179261915075
```

Note that the number of iterations to be performed is computed when the iterator is evaluated before the loop is started. Changing the value of the upper bound in the iterator from inside the loop will, therefore, not change the number of iterations to be performed. The start, stop, and increment values can be general expressions. The only condition is that the number of iterations to perform evaluates to a number (other than a complex number). The increment need not divide evenly into the interval from start to stop. The number of iterations in the loop `Do[`*body*`, {`*start*`,` *stop*`,` *incr*`}]` is $(stop - start)/incr + 1$, rounded down to the nearest integer.

The number of steps is `(4a - 2a)/a + 1` ⟶ 3, an integer.

```
In[2]:= Do[ Print[i], {i, 2a, 4a, a} ]

2 a
3 a
4 a
```

The number of steps, `(3.5 - 0.0)/1 + 1` ⟶ 4.5, is rounded to 4.

```
In[4]:= Do[ Print[r], {r, 0.0, 3.5} ]

0.
1.
2.
3.
```

In a nested loop, the iterator for the inner variable `j` is evaluated for each value of the outer variable `i`.

```
In[6]:= Do[ Print[{i, j}], {i, 3}, {j, i} ]

{1, 1}
{2, 1}
{2, 2}
{3, 1}
{3, 2}
{3, 3}
```

`Do[]` in *Mathematica* is similar to the `DO` loop in FORTRAN or BASIC and to the `for` loop in PASCAL.

■ 4.2.2 Conditional Repetition of Statements

Often we want to perform a calculation repeatedly while a certain condition is true and stop as soon it becomes false. In this case, we do not know the number of iterations that are to be performed in advance and cannot use the `Do[]` loop, but use the `While[]` loop instead. Its form is `While[`*condition*`,` *body*`]`. Before each iteration the *condition* is tested. If the test returns `True`, the body of the loop is evaluated one more time, otherwise the loop terminates without returning a value. If the test does not return `True` the first time it is tested, the loop body is not executed at all.

For an example we look at the function $\pi(x)$ that, given a number x as argument, finds the number of primes less than x. Prime numbers are positive integers that have no divisors except 1 and themselves. The sequence starts with 2, 3, 5, 7, 11, ... The function `Prime[`n`]` returns the n^{th} prime number p_n. We use iteration to find an n so that $p_n \leq x < p_{n+1}$. To avoid name conflicts with the built-in function `PrimePi[]` we call our version `primePi[]`, see Listing 4.2–2.

```
Attributes[primePi] = {Listable}

primePi[x_/; x < 2] := 0

primePi[x_/; x >= 2] :=
    Module[{li = LogIntegral[x], n0, n1, m},
        n0 = Floor[li - LogIntegral[Sqrt[x]]];
        n1 = Ceiling[li];
        While[ n1 - n0 > 1,
            m = Floor[(n0 + n1)/2];              (* midpoint *)
            If[ Prime[m] <= x, n0 = m, n1 = m ]
        ];
        n0
    ]
```

Listing 4.2–2: PrimePi.m: Find the index of a prime number

The function primePi[] first gets an initial guess for n. It is a famous theorem in mathematics that the logarithmic integral gives a rather good guess for n. In the range in which Prime[] is implemented this guess is always too large. On the other hand, subtracting the logarithmic integral of $\sqrt{x}$ gives an estimate that is too low. Therefore, we maintain two variables n_0 and n_1 that bracket the correct value and use bisection to cut the interval that must contain the correct value in half. At the beginning, we know that the n we are looking for lies between n_0 and n_1. At each iteration, we check whether the midpoint between them is too high or too low. If it is too low, we set n_0 to this midpoint otherwise we set n_1 to it. In each iteration, the interval between n_0 and n_1 is cut in half and after a few steps the two will differ by at most 1 and we have found n.

Our algorithm does not work for arguments smaller than 2. We give a separate rule for this case. There are no primes smaller than 2 and so the value of primePi[] is 0 in this case. (The built-in function PrimePi[] works similarly to ours but uses a more sophisticated guess to start the iteration.)

The number 1997 is the 302$^{\text{nd}}$ prime.

```
In[1]:= primePi[1997]
Out[1]= 302
```

Here is a simple test.

```
In[2]:= Prime[ % ]
Out[2]= 1997
```

This computes the $100,000^{\text{th}}$ prime.

```
In[3]:= Prime[100000]
Out[3]= 1299709
```

A second test of our function.

```
In[4]:= primePi[ % + 1 ]
Out[4]= 100000
```

Our function works also for exact numeric quantities. There is no need to convert them into approximate numbers first; see Section 7.3.

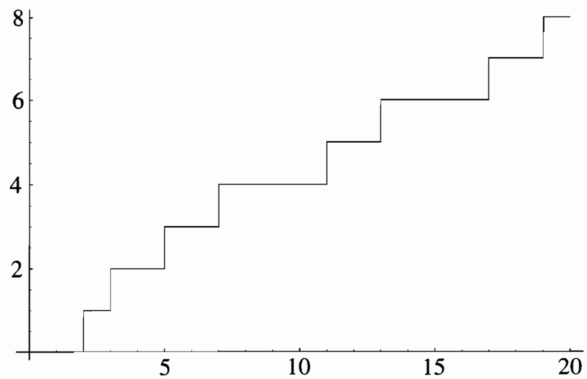
```
In[5]:= PrimePi[2^10 Pi]
Out[5]= 454
```

primePi[] is listable and it can therefore compute $\pi(x)$ for a whole list of numbers in one call.

```
In[6]:= primePi[ Range[10] ]
Out[6]= {0, 1, 2, 2, 3, 3, 4, 4, 4, 4}
```

primePi[] is defined for all real numbers. We can, therefore, even plot it. The function jumps by one at primes and is constant in between.

```
In[7]:= Plot[ primePi[x], {x, 0, 20} ];
```



While[] in *Mathematica* is similar to the while loop in PASCAL and **C**.

■ 4.2.3 The For Loop

The For[] loop is patterned after the corresponding loop in the language **C**. It does not have an equivalent in other languages. **C** lacks the equivalent of the Do[] loop and so expert **C** programmers often end up using For[] instead of Do[] to iterate over the values of a variable in *Mathematica*. Instead of Do[*body*, {*var*, *start*, *stop*}], they write For[*var*=*start*, *var*<=*stop*, *var*++, *body*], which looks a bit clumsy. The For[] loop is useful in more complicated instances where the iteration involves several variables and end conditions. The For[] loop can easily be expressed in terms of a While[] loop. Instead of

For[*start*, *test*, *step*, *body*]

we could write

start; While[*test*, *body*; *step*].

If you are not familiar with For[], this correspondence can help you understand how it works.

■ 4.2.4 Testing the Exit Condition at the End of a Loop

Besides the While[] loop that checks a condition at the beginning of each iteration, many programming languages offer also a loop that checks the condition *after* each iteration. Here are several ways in which we could implement a hypothetical Until[*body*, *test*] that repeats *body* until *test* becomes True.

`While[True, `*body*`; If[`*test*`,Break[]]]`	break from an infinite `While[]` as soon as *test* is true.
`t=False; While[ !t, `*body*`; t=`*test* `]`	remember the value of *test* for the next iteration.
`For[ t=False, !t, t=`*test*`, `*body* `]`	use a `For[]` loop.

Loops that check their test at the end of an iteration

If we wanted, we could define our own command `Until[]` in terms of one of the possibilities outlined. The minipackage Until.m is shown in Listing 4.2–3.

```
BeginPackage["ProgrammingInMathematica`Until`"]

Until::usage = "Until[body, test] evaluates body until test becomes true."

Begin["`Private`"]

Attributes[Until] = {HoldAll}

Until[body_, test_] := Module[ {t}, For[ t=False, !t, t=test, body ] ]

End[]

EndPackage[]
```

Listing 4.2–3: Until.m: A loop that checks its test after each iteration

The attribute `HoldAll` prevents evaluation of the parameters of `Until[]`. They are evaluated only inside `For[]`. All loops should behave like this.

This is a fixed point iteration. It sets x to $1 + 1/x$ until x is equal to $1 + 1/x$ (correct to machine precision). This number is called the *golden ratio.* More about fixed points can be found in Section 4.4.1. Note the difference between the use of `=` and `==`.

```
In[1]:= x = 1.0;\
        Until[ x = 1 + 1/x, x == 1 + 1/x ]; x
Out[1]= 1.61803
```

`Until[]` is similar to `repeat...until` in PASCAL and to `do...while` in **C** (with the truth value of the test reversed!).

4.3 Structured Iteration

In many programming languages the loops presented in Section 4.2 are all that is available. The structured iteration commands of this section are perhaps new to you. If you have programmed in LISP or APL, then you will, however, recognize many familiar and useful commands. The flow-control statements of traditional languages were not designed with the applications in mind that are now possible in *Mathematica*, but rather they were selected for ease of implementation and the need of applications in computer science itself.

Because *Mathematica* does offer the traditional looping constructs, as we have just seen, it is rather tempting to simply continue using these familiar means of flow control instead of using a more natural, problem-oriented approach. In this section, I would like to show you the transformation from the old approach to *Mathematica*'s way of functional programming.

4.3.1 Sums and Products

Given the problem of adding the square roots of the first 500 integers, the solution in most programming languages is to use an auxiliary variable that is incremented by the square root of a loop index iterating from 1 to 500.

```
sum = 0.0;
Do[ sum = sum + N[Sqrt[i]], {i, 1, 500} ];
sum
```

The procedural way of adding numbers

In *Mathematica*, this loop reduces to a single statement that directly corresponds to the mathematical formula $\sum_{i=1}^{500} \sqrt{i}$.

```
Sum[ N[Sqrt[i]], {i, 1, 500} ]
```

The mathematical way of adding numbers

Mathematica does not force you to think about *how* to implement a summation, but lets you focus on the concept itself instead. `Product[]` works in the same way, multiplying its terms together instead of adding them up.

This computes the product $\prod_{i=0}^{5}(x-i)$ and expands it.

```
In[1]:= Product[ x-i, {i, 0, 5} ] // Expand

                       2        3       4       5    6
Out[1]= -120 x + 274 x  - 225 x  + 85 x  - 15 x  + x
```

The iterator used for sums and products is the same as that used for loops (see Section 4.2.1). This allows a rather sophisticated way of computing the same product.

```
In[2]:= Product[ e, {e, x, x-5, -1} ] // Expand

                       2        3       4       5    6
Out[2]= -120 x + 274 x  - 225 x  + 85 x  - 15 x  + x
```

■ 4.3.2 Tables

`Sum[]` and `Product[]` are two examples of a class of commands that evaluate an expression several times varying one or several index variables and then collecting the results in a specific way, either adding them up or multiplying them together. The `Table[]` command is the simplest of them. It just collects its results in a list.

This generates a list of the first 10 powers of x.

```
In[1]:= Table[ x^i, {i, 0, 9} ]

                 2   3   4   5   6   7   8   9
Out[1]= {1, x, x , x , x , x , x , x , x , x }
```

This generates a list of the first six of the polynomials used in the previous example. Note how the inner index (in the product) depends on the outer index (of the table).

```
In[2]:= Table[ Product[x-j, {j, 0, i}], {i, 0, 5} ]

Out[2]= {x, (-1 + x) x, (-2 + x) (-1 + x) x,

  (-3 + x) (-2 + x) (-1 + x) x,

  (-4 + x) (-3 + x) (-2 + x) (-1 + x) x,

  (-5 + x) (-4 + x) (-3 + x) (-2 + x) (-1 + x) x}
```

`TableForm[]` prints the elements of a list on separate lines as a table.

```
In[3]:= TableForm[%]

Out[3]//TableForm=

 x

 (-1 + x) x

 (-2 + x) (-1 + x) x

 (-3 + x) (-2 + x) (-1 + x) x

 (-4 + x) (-3 + x) (-2 + x) (-1 + x) x

 (-5 + x) (-4 + x) (-3 + x) (-2 + x) (-1 + x) x
```

`Table[]`, like all of these structured iterators, can have more than one iterator specification, giving you multidimensional tables. Two iterators generate matrices. The first iterator is the outermost one. `Table[`*expr*`,` *iterator*$_1$`,` *iterator*$_2$`]` is therefore equivalent to `Table[Table[`*expr*`,` *iterator*$_2$`],` *iterator*$_1$`]`.

The matrix of the first n powers of n different variables is called *Vandermonde*'s matrix.

```
In[4]:= Table[ x[i]^j, {i, 1, 5}, {j, 0, 4} ] // MatrixForm

                                   2       3       4
Out[4]//MatrixForm=  1    x[1]    x[1]    x[1]    x[1]

                                   2       3       4
                     1    x[2]    x[2]    x[2]    x[2]

                                   2       3       4
                     1    x[3]    x[3]    x[3]    x[3]

                                   2       3       4
                     1    x[4]    x[4]    x[4]    x[4]

                                   2       3       4
                     1    x[5]    x[5]    x[5]    x[5]
```

Its determinant in expanded form has $n!$ terms. Here n is 5, and we get 120 terms—too much to print out in full.

```
In[5]:= Det[%] // Short

Out[5]//Short=

     4      3      2                              2      3      4
 x[1]  x[2]  x[3]  x[4] + <<118>> + x[2] x[3]  x[4]  x[5]
```

But it factors in this nice way.

```
In[6]:= Factor[%]
Out[6]= (x[1] - x[2]) (x[1] - x[3]) (x[2] - x[3])
  (x[1] - x[4]) (x[2] - x[4]) (x[3] - x[4]) (x[1] - x[5])
  (x[2] - x[5]) (x[3] - x[5]) (x[4] - x[5])
```

■ 4.3.3 Arrays

To understand the difference between `Array[]` and the other structured iterators we have discussed so far, let us have a closer look at how these iterators work. Assume the iterator `Table[`*expr*`, {i, `*start*`, `*final*`}]`, where the iterator variable is `i`. First, the variable is set to the initial value, *start*. This is done in much the same way as if you had typed `i=`*start*. Then, *expr* is evaluated, using the current value for `i` wherever `i` occurs in *expr*. For the next iteration, the next value *start*`+1` is assigned to `i` and *expr* is evaluated again.

Another way to look at this is to say that the expression *expr* describes a function of the iterator variable `i`. For each value of `i`, we get a value *f*`[i]` for some function *f*. The iterator `Array[]` takes this point of view. In `Array[`*f*`, `*n*`]`, *f* is a function that is applied to each value of the iterator in turn. We do not need a named iterator variable, as the name of the parameter of a function does not matter at all. *Mathematica* simply generates the expressions *f*`[1]`, *f*`[2]`, ..., *f*`[`*n*`]` and evaluates them.

You could express this action using `Table[]`: The expression `Array[`*f*`, `*n*`]` is equivalent to `Table[`*f*`[i], {i, `*n*`}]` (assuming that there is no conflict of variable names, that is, `i` does not appear as a free variable in the definition of *f*).

This generates a list of the first 20 prime numbers.

```
In[1]:= Array[ Prime, 20 ]
Out[1]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
  43, 47, 53, 59, 61, 67, 71}
```

Here is the equivalent `Table[]` command.

```
In[2]:= Table[ Prime[i], {i, 20} ]
Out[2]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
  43, 47, 53, 59, 61, 67, 71}
```

You can also express a `Table[]` in terms of an `Array[]`. If the expression to be tabulated is not of the form *f*`[`*var*`]`, where *var* is the iterator variable, you need to express it as a pure function. For example, `Table[i^2, {i, 10}]` can be expressed as `Array[Function[i, i^2], 10]` (for more on pure functions, see Section 5.2). In general, you can always write `Table[`*expr*`, {`*var*`, `*n*`}]` as `Array[Function[`*var*`, `*expr*`], `*n*`]`.

`Array[]` is not as flexible as `Table[]`. The increment of the iterator is always 1. The iterator starts at 1 unless a different initial value is given as a third argument, as in `Array[f, 3, 0]` ⟶ `{f[0], f[1], f[2]}`. A multidimensional array has the form `Array[`*f*`, {`n_1`, `n_2`, ..., `n_k`}]` and corresponds to the table

`Table[` *f*`[`i_1`, `i_2`, ..., `i_n`], {`i_1`, `n_1`}, {`i_2`, `n_2`}, ..., {`i_k`, `n_k`} ]`.

This array gives again Vandermonde's matrix from Section 4.3.2. We have two iterators and so we need a pure function of two variables.

```
In[4]:= Array[ x[#1]^(#2-1)&, {5, 5} ] // MatrixForm
Out[4]//MatrixForm=
                          2       3       4
        1     x[1]    x[1]    x[1]    x[1]

                          2       3       4
        1     x[2]    x[2]    x[2]    x[2]

                          2       3       4
        1     x[3]    x[3]    x[3]    x[3]

                          2       3       4
        1     x[4]    x[4]    x[4]    x[4]

                          2       3       4
        1     x[5]    x[5]    x[5]    x[5]
```

■ 4.3.4 Mapping a Function over a List

Quite often we want to apply some function to a list of expressions. Consider the example of squaring all the numbers in a list. A procedural program would iterate over the list and build up a new list of the results:

```
SquareList[l_List] :=
    Module[{result = {}, i},
        Do[ AppendTo[result, l[[i]]^2], {i, Length[l]} ];
        result
    ]
```

Using a loop to apply a function to the elements of a list

Building up the result by using `Append[]` or `AppendTo[]` when you know the length of the result beforehand is not a good solution. It is similar to using an auxiliary variable for adding up the terms in a sum (see Section 4.3.1). In the last section, we saw how to improve such iterations by using a structured iterator. Because we want a list of the results, we use `Table[]`:

```
SquareList[l_List] :=
    Module[{i},
        Table[ l[[i]]^2, {i, Length[l]} ]
    ]
```

Using a structured iterator to apply a function to the elements of a list

In each of these cases, we apply the same function to each element of the original list. In such cases we should use `Map[`*f*`, `*list*`]` to perform the operation. `Map[]` takes the name of a function *f* and applies it to all the elements of *list*. The function we want is "square the argument." There is no built-in function for this, so we use a pure function instead. The pure function that squares its argument can be written as `Function[e, e^2]`.

```
SquareList[l_List] := Map[ Function[e, e^2], l ]
```

Mapping a function over the elements of a list

This operation is so common that most of the built-in functions map themselves automatically over lists. This property is called *listability*. Taking advantage of this, our example becomes even simpler.

```
SquareList[l_List] := l^2
```

Using the built-in listability of `Power[]`

■ 4.3.5 Listability

It is worth understanding exactly what happens if a listable function has more than one argument. The internal form of `l^2` is `Power[l, 2]`. Let us assume that the value of `l` is `{a, b, c}`, and so our expression is `Power[{a, b, c}, 2]`. The first argument of `Power[]` is a list, while its second argument is a number. In this case, the expression is transformed into a list of powers, duplicating the second argument, and we get `{Power[a, 2], Power[b, 2], Power[c, 2]}`, or `{a^2, b^2, c^2}`. If all the arguments of a listable function are lists, then their elements are picked in parallel. `Power[{a, b, c}, {1, 2, 3}]`, or `{a, b, c}^{1, 2, 3}` in the usual notation, becomes `{Power[a, 1], Power[b, 2], Power[c, 3]}`, or `{a, b^2, c^3}`. In this case, all the argument lists must have the same length.

The second argument of `Plus` is repeated as often as necessary.

```
In[1]:= {a, b, c} + 1
Out[1]= {1 + a, 1 + b, 1 + c}
```

The elements of the two lists are multiplied in sequence and the list of results is returned.

```
In[2]:= {1, 2, 3} {x, y, z}
Out[2]= {x, 2 y, 3 z}
```

You can declare your own functions to be listable by giving them the attribute `Listable`. We have already given an example of this in the function `primePi` in Section 4.2.2.

Addition is listable, because it has the attribute `Listable`.

```
In[3]:= Attributes[ Plus ]
Out[3]= {Flat, Listable, NumericFunction, OneIdentity,
  Orderless, Protected}
```

The function `f` is declared as listable. Such a definition should come before any rules defined for the function.

```
In[4]:= SetAttributes[ f, Listable ]
```

The function behaves as explained above.

```
In[5]:= f[ {a, b, c}, x ]
Out[5]= {f[a, x], f[b, x], f[c, x]}
```

Some functions behave like listable ones but they do not have the attribute `Listable`. The reason is that sometimes a list as an argument has a special meaning. Consider differentiation. The general form is `D[`*expr*`, {`*var*`, `*n*`}]` to differentiate *expr* n times with respect to *var*. The list in the second parameter has a special meaning. If the first argument is a list, then we still want each of its elements to be differentiated n times

with respect to *var*. If `D[]` had the attribute `Listable`, then `D[{e1, e2}, {x, 3}]` would turn into `{D[e1, x], D[e2, 3]}`, which does not make sense. What we want is `{D[e1, {x, 3}], D[e2, {x, 3}]}`.

You can give a simple rule for one of your own functions to make it behave in the same way.

```
f[ l_List, args___ ] := Map[ f[#, args]&, 1 ]
f[ e_, ... ] := ...   (* code for the usual case *)
```

A function that is listable over its first argument only

This rule works for functions of any number of additional arguments because we used the *triple blank* pattern `arg___` to match any number of elements.

The following example defines rules for `diff[`*expr*`,` var_1`,` var_2`, ...,` var_n`]`, a function that differentiates its first argument once with respect to each of the variables given as additional arguments.

Make `diff` listable over its first argument.

```
In[1]:= diff[l_List, args___] := Map[ diff[#, args]&, 1 ]
```

Differentiate once with respect to the first variable.

```
In[2]:= diff[e_, var_, rest___] := diff[ D[e, var], rest ]
```

No differentiation, if no variables are given.

```
In[3]:= diff[e_] := e
```

These rules are all it takes. As an example, we differentiate the three functions `x`, `x^y`, and `x y` with respect to `x` and then with respect to `y`.

```
In[4]:= diff[{x, x^y, x y}, x, y]

             -1 + y    -1 + y
Out[4]= {0, x       + x       y Log[x], 1}
```

The rules we have given are, of course, none other than the ones already built into `D[]`.

```
In[5]:= D[{x, x^y, x y}, x, y]

             -1 + y    -1 + y
Out[5]= {0, x       + x       y Log[x], 1}
```

4.4 Iterated Function Application

Often, explicit loops are only a poor disguise for higher-level operations: iterated and nested application of functions. *Mathematica* provides all tools to express these higher constructs directly, without clumsy detours.

4.4.1 Fixed-Point Iteration

A common task is to repeatedly apply a function to an expression. An example is Newton's iteration formula for approximating the square root of a number r. Given an approximation x_i, we can find a better approximation by computing $x_{i+1} = (x_i + r/x_i)/2$. Starting with an initial guess x_0 we get a sequence of better and better approximations $x_1, x_2, \ldots$ that converges to $\sqrt{r}$.

We want to compute an approximation to $\sqrt{2}$ starting with the initial guess 1. We want the computations to be accurate to 40 digits.

```
In[1]:= r = 2; x = N[1, 40];
```

This computes and prints the first seven iterations.

```
In[2]:= Do[ x = (x + r/x)/2; Print[x], {7}]

1.500000000000000000000000000000000000000
1.416666666666666666666666666666666666667
1.414215686274509803921568627450980392157
1.414213562374689910626295578890134910117
1.414213562373095048801689623502530243615
1.414213562373095048801688724209698078570
1.414213562373095048801688724209698078570
```

The last value of `x` is accurate within the precision of 40 digits.

```
In[3]:= x - N[Sqrt[r], 40]

              -40
Out[3]= 0. 10
```

There is a more elegant way to express this iteration. The iteration is of the form `x = f[x]`, with $f\colon x \mapsto (x + r/x)/2$. The function `Nest[f, x, n]` allows us to apply a function f a given number of times n to an initial value x. Using this, we do not need to program a loop as we just did. We need to express the formula for the new x as a function of the old x. The pure function `Function[x, (x + r/x)/2]` is a straightforward translation of f into *Mathematica*.

Initialize r.

```
In[4]:= r = 2;
```

This command performs seven iterations and returns the last result. It is the same as in the previous calculation.

```
In[5]:= Nest[ Function[x, (x + r/x)/2], N[1, 40],
              7
        ]

Out[5]= 1.414213562373095048801688724209698078570
```

How many iterations are necessary to guarantee accuracy to the number of digits that we have specified? One way to find out when to stop the iteration is to compare the result of the last iteration with the previous value of x and stop if that difference gets smaller than the precision with which we do the calculations. *Mathematica* can detect this condition by itself. The function `FixedPoint[`f`, `x`]` keeps applying f to x just as `Nest[]` does, but it stops as soon as the expression no longer changes. A value x so that $f(x)$ is equal to x is called a *fixed point* of f.

Initialize r.

```
In[6]:= r = 2;
```

This performs as many iterations as are necessary to find the fixed point of `Function[x, (x + r/x)/2]` and returns the last result.

```
In[7]:= FixedPoint[ Function[x, (x + r/x)/2], N[1, 40] ]
Out[7]= 1.414213562373095048801688724209698078570
```

`FixedPointList[]` returns all intermediate results. In this way we can find out the intermediate values and can count the number of iterations performed.

```
In[8]:= FixedPointList[ Function[x, (x + r/x)/2],
                        N[1, 40] ] // TableForm
Out[8]//TableForm=
 1.000000000000000000000000000000000000000
 1.500000000000000000000000000000000000000
 1.416666666666666666666666666666666666667
 1.414215686274509803921568627450980392157
 1.414213562374689910626295578890134910117
 1.414213562373095048801689623502530243615
 1.414213562373095048801688724209698078570
 1.414213562373095048801688724209698078570
```

The cosine function also has a fixed point. This iteration is however much slower because the rate of convergence is smaller than in Newton's iteration.

```
In[9]:= FixedPoint[ Cos, 0.5 ]
Out[9]= 0.739085
```

The cosine of the last result is in fact equal to it.

```
In[10]:= Cos[%] == %
Out[10]= True
```

You have to be careful when using `FixedPoint[]`. If a function does not have a fixed point, or if the iteration does not converge, *Mathematica* gets trapped in an infinite loop. If this happens, you can interrupt and abort the calculation. As a precaution, you can give a third argument to `FixedPoint[]` that specifies the maximum number of iterations to perform.

■ 4.4.2 Application: Newton's Iteration Formula in General

In the preceding subsection we have briefly alluded to Newton's iteration formula for finding square roots of functions. The method can be used more generally to find zeroes

of functions. A zero of a function is, of course, a number x so that $f(x) = 0$. The Newton iteration proceeds by finding better and better approximations to a zero of f. Given an approximation x_i we find a better one with the formula $x_{i+1} = x_i - f(x_i)/f'(x_i)$.

`NewtonZero[`*f*`,` *start*`]`	find a zero of the function *f*
`NewtonZero[`*expr*`,` *var*`,` *start*`]`	find a zero of *expr* as a function of *var*
`NewtonFixedPoint[`*f*`,` *start*`]`	find a fixed point of the function *f*
`NewtonFixedPoint[`*expr*`,` *var*`,` *start*`]`	find a fixed point of *expr* as a function of *var*

Zeroes and fixed points of functions using Newton's method

The package Newton1.m, shown in Listing 4.4–1, is a preliminary version of Newton.m. It defines the two functions `NewtonZero[]` and `NewtonFixedPoint[]`, which find zeroes and fixed points of functions, respectively. `NewtonZero[]` implements Newton's iteration formula. It is called as `NewtonZero[`*expr*`,` *var*`,` *start*`]`, in which you give an expression *expr* involving the variable *var* and an initial point *start* for the iteration. It can also be called as `NewtonZero[`*f*`,` *start*`]`, in which you give the name *f* of a function and again an initial point *start* for the iteration. The command `NewtonFixedPoint[`*f*`,` *start*`]` finds a fixed point of the function *f* starting the iteration at *start*.

`NewtonZero[]` performs up to `$RecursionLimit` many iterations to find a zero of the given function. The maximal number of iterations can be changed with the option `MaxIterations`, which is used in several built-in commands for the same purpose. If it cannot find a zero within the precision of the initial point given, it prints a message and returns the value found so far.

Let us look at the code for `NewtonZero[f_, x0_]`. After dealing with the options in the usual way, it first finds the precision of `x0` and locally sets `$MaxPrecision` to this value (plus some extra precision). As a consequence, the following fixed-point computation will never use numbers with a precision higher than the input. (Fixed-point computations with arbitrary-precision numbers can run away, as we shall see in Section 7.2.4.) Then, it computes the derivative `f'` of the given function `f`. The value thus computed is bound in `With[]` so that it will not be recomputed at each iteration step. Next, we find the fixed point of the pure function `(# - f[#]/fp[#])&`, which corresponds to Newton's formula $x_i - f(x_i)/f'(x_i)$. After returning from the fixed-point calculation, we compute `f[res]`, which will be close to zero if we did in fact find the fixed point. If `Abs[f[res]] < 10^-prec`, we found the zero at least to precision `prec`, the input precision. If this condition is not satisfied we print a message.

If `NewtonZero[]` is called in the form `NewtonZero[`*expr*`,` *var*`,` *start*`]`, the function whose zero we want to find is given as an expression *expr* that is to be considered a function of *var*. We can turn it into the pure function `Function[`*var*`,` *expr*`]` and then simply use the other definition.

To find a fixed point of the function $f(x)$, we find a zero of the function $f(x) - x$. `NewtonFixedPoint[`*f*`,` *start*`]`, therefore, simply calls `NewtonZero[]` with the pure

```
BeginPackage["ProgrammingInMathematica`Newton`"]

NewtonZero::usage = "NewtonZero[f, x0] finds a zero of the function f using
    the initial guess x0 to start the iteration. NewtonZero[expr, x, x0]
    finds a zero of expr as a function of x. The recursion limit
    determines the maximum number of iteration steps that are performed."

NewtonFixedPoint::usage = "NewtonFixedPoint[f, x0] finds a fixed point of the
    function f using the initial guess x0 to start the iteration.
    NewtonFixedPoint[expr, x, x0] finds a fixed point of expr as a function of x."

Options[NewtonZero] = Options[NewtonFixedPoint] = {
    MaxIterations :> $RecursionLimit
}

Newton::noconv = "Iteration did not converge in `1` steps."

Begin["`Private`"]

extraPrecision = 10 (* the extra working precision *)

NewtonZero[ f_, x0_, opts___?OptionQ ] :=
    Module[{res, maxiter},
      maxiter = MaxIterations /. {opts} /. Options[NewtonZero];
      With[{prec = Precision[x0], fp = f'},
        Block[{$MaxPrecision = prec + extraPrecision},
            res = FixedPoint[(# - f[#]/fp[#])&, x0, maxiter]
        ];
        If [ !TrueQ[Abs[f[res]] <= 10^-prec],
             Message[Newton::noconv, maxiter] ];
        res
      ]
    ]

NewtonZero[ expr_, x_, x0_, opts___?OptionQ ] :=
    NewtonZero[ Function[x, expr], x0, opts ]

NewtonFixedPoint[ f_, x0_, opts___?OptionQ ] :=
    Module[{maxiter},
        maxiter = MaxIterations /. {opts} /. Options[NewtonFixedPoint];
        NewtonZero[(f[#] - #)&, x0, MaxIterations -> maxiter]
    ]

NewtonFixedPoint[ expr_, x_, x0_, opts___?OptionQ ] :=
    NewtonFixedPoint[ Function[x, expr], x0, opts ]

End[ ]

Protect[ NewtonZero, NewtonFixedPoint ]

EndPackage[ ]
```

Listing 4.4–1: Newton1.m: Newton's iteration formula

function `(`*f*`[#] - #)&`. Note that we do deal with the option `MaxIterations` before calling `NewtonZero[]`. This is done so that the correct default, namely the one defined for `NewtonFixedPoint`, is used, not the one defined for `NewtonZero`.

If you call `NewtonFixedPoint[`*f*`,` *start*`]` with a pure function *f* itself, the program goes through many levels of constructing new pure function from old ones. Nevertheless, it is capable of differentiating them correctly.

This gives again the fixed point of the cosine function, but much faster than in Section 4.4.1 on page 95.

```
In[1]:= NewtonFixedPoint[ Cos, N[1, 40] ]
Out[1]= 0.7390851332151606416553120876738734040
```

Here is the golden ratio to 20 digits as fixed point of the function $x \mapsto 1 + 1/x$, much faster than in Section 4.2.4 on page 86.

```
In[2]:= NewtonFixedPoint[ 1 + 1/x, x, N[2, 20] ]
Out[2]= 1.61803398874989485
```

The function $z^2 + 1$ does not have any real zeroes and so the iteration cannot converge.

```
In[3]:= NewtonZero[ z^2 + 1, z, 0.51 ]
Newton::noconv: Iteration did not converge in 256 steps.
Out[3]= 1.31636
```

If we give a complex initial point, however, it converges to one of the two complex zeroes.

```
In[4]:= NewtonZero[ z^2 + 1, z, 0.5 + I ]
Out[4]= 0. + 1. I
```

An initial point with a negative imaginary part converges to the other complex zero.

```
In[5]:= NewtonZero[ z^2 + 1, z, 0.5 - I ]
Out[5]= 0. - 1. I
```

The built-in function `FindRoot[]` use similar techniques to find numerical solutions to equations.

```
In[6]:= FindRoot[ z^2 + 1 == 0, {z, 0.5+I}]
                              -10
Out[6]= {z -> 2.01241 10    + 1. I}
```

If the initial guess `x0` is an exact number and the function whose fixed point we want to find returns an exact result for exact arguments, we may never find the fixed point, but we can still find a good rational approximation, as this amusing example shows.

We perform at most five steps in an attempt to find the exact fixed point, the value of the golden ratio. The result is a rational approximation of `GoldenRatio`.

```
In[7]:= NewtonFixedPoint[1 + 1/x, x, 2, MaxIterations -> 5]
Newton::noconv: Iteration did not converge in 5 steps.
        51680708854858323072
Out[7]= --------------------
        31940434634990099905
```

The approximation is correct to almost 40 decimal digits.

```
In[8]:= N[-Log[10, Abs[GoldenRatio - %]], 20]
Out[8]= 39.35816664532795001
```

The final package Newton.m contains additional code for better control of numerical accuracy. It is described in Section 7.2.4.

■ 4.4.3 Nesting Functions of One Variable

The operation `Nest[`f`, `x_0`, `n`]` implements iteration of a function of one variable, computing the sequence

$$x_i = f(x_{i-1}), \quad i = 1, \ldots, n.$$

`NestList[`f`, `x_0`, `n`]` returns a list of all x_i.

This symbolic example shows how it works.

```
In[1]:= NestList[ f, x, 4 ]
Out[1]= {x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

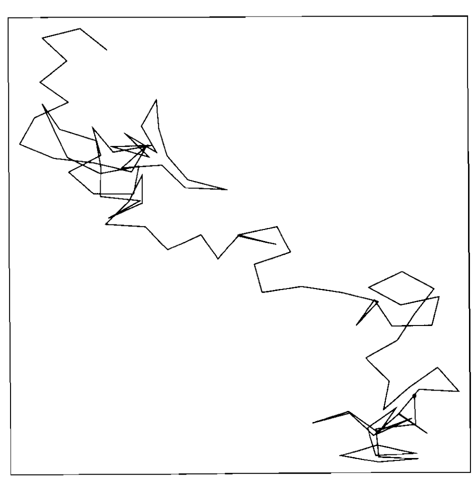
A random walk can easily be generated with `NestList[]`. The function to nest takes a point as argument and adds a random displacement. The result returned from `NestList[]` is the trail of the random walk, that is, a list of all positions visited. Listing 4.4–2 shows a simple package for random walks. The auxiliary variable `randomDelta` returns a random unit vector; note that it must be defined with `:=`, not with `=`, because the right side needs to be evaluated anew every time the variable is used (to obtain a fresh random number).

```
BeginPackage["ProgrammingInMathematica`RandomWalk`"]

RandomWalk::usage = "RandomWalk[n, opts] plots a random walk of length n.
    Options of Graphics are passed to Show."

Begin["`Private`"]

range = N[{0, 2Pi}]

randomDelta :=
    With[{dir = Random[Real, range]}, {Cos[dir], Sin[dir]}]

RandomWalk[n_Integer, opts___?OptionQ] :=
    Module[{points},
        points = NestList[ # + randomDelta&, {0, 0}, n ];
        Show[ Graphics[{Point[{0, 0}], Line[points]}], opts,
              Frame -> True, FrameTicks -> None, AspectRatio -> 1
        ]
    ]

End[]

Protect[ RandomWalk ]

EndPackage[]
```

Listing 4.4–2: RandomWalk.m: Random walks in two dimensions

Here is a random walk with 100 steps. For a larger example, see the chapter opener picture on page 335.

```
In[1]:= RandomWalk[100];
```



4.4.4 Folding Functions of Two Variables

An iteration similar to `Nest[]` can be defined for a function g of two variables. One of the arguments is filled with the result of the previous iteration, the other argument is taken from a list of values, that is, we iterate

$$x_i = g(x_{i-1}, y_i), \quad i = 1, \dots, n. \tag{4.4–1}$$

This operation is available as `Fold[`g`, `x_0`, {`y_1`, ..., `y_n`}]`. With `FoldList[]` instead of `Fold[]`, we get the list of all intermediate results x_i, $i = 0, 1, \dots, n$, as expected.

The results of `NestList[`f`, `x_0`, `n`]` and `FoldList[`g`, `x_0`, {`y_1`, ..., `y_n`}]` are lists with $n + 1$ elements, not with n elements, as one might expect.

Again, a symbolic example shows clearly how the operation works.

```
In[2]:= Fold[g, x0, {y1, y2, y3}]
Out[2]= g[g[g[x0, y1], y2], y3]
```

Here, we see all intermediate results.

```
In[3]:= FoldList[g, x0, {y1, y2, y3}]
Out[3]= {x0, g[x0, y1], g[g[x0, y1], y2],
  g[g[g[x0, y1], y2], y3]}
```

4.4.4.1 Folding Associative Operations and Prefix Sums

If you choose x_0 as the neutral element of an associative binary operation g (that is, $g(x_0, y) = y$), folding implements the extension of this operation to several arguments through the formula

$$g(y_1, y_2, \dots, y_n) = g(\dots g(g(x_0, y_1), y_2), \dots, y_n).$$

The number 0 is the neutral element of addition; therefore, this computation gives the sum of the elements of `{a, b, c, d}`. Of course, the built-in `Plus` extends automatically to several arguments, so this construction is not necessary here.

```
In[4]:= Fold[ Plus, 0, {a, b, c, d} ]
Out[4]= a + b + c + d
```

This form, however, is sometimes useful. It computes the *prefix sums* of a list.

```
In[5]:= FoldList[ Plus, 0, {a, b, c, d} ]
Out[5]= {0, a, a + b, a + b + c, a + b + c + d}
```

One application of prefix sums is the computation of discrete cumulative probabilities. Assume you want to draw a random integer from 1 to n, where the number k is drawn with probability p_k, and $\sum_{k=1}^{n} p_k = 1$. One approach is to compute the cumulative probabilities x_i with

$$x_i = \sum_{k=1}^{i} p_k, \quad i = 1, \dots, n.$$

To find a random i with the desired distribution, first find a uniformly distributed random real number r with $0 \leq r \leq 1$ and then determine the smallest i with $x_i \geq r$. Here is a small function RandomDistributed[{p_1, ..., p_n}] that implements this idea (see Section 12.2.3 for an application):

```
RandomDistributed[p_List] :=
    Module[{x, r},
        x = FoldList[ Plus, 0, p ];
        x = Drop[x, 1];
        r = Random[];
        Position[x, xi_ /; xi >= r, {1}, 1][[1,1]]
    ]
```

Random integers with given probability distribution

Let us see how it works. Here is a list of four probabilities summing up to 1.

```
In[6]:= p = {0.1, 0.4, 0.3, 0.2};
```

Here are the cumulative probabilities.

```
In[7]:= x = FoldList[ Plus, 0, p ]
Out[7]= {0, 0.1, 0.5, 0.8, 1.}
```

We need to remove the first trivial entry.

```
In[8]:= x = Drop[x, 1]
Out[8]= {0.1, 0.5, 0.8, 1.}
```

Here is a random number, uniform between 0 and 1.

```
In[9]:= r = Random[]
Out[9]= 0.194288
```

We find the position of the first entry of x that is larger than r. The third argument {1} of Position[] restricts the search to the first level of x. This restriction is not really necessary here, but we need to have something there, because we want to specify the fourth argument 1 to find just *one* position. Usually, Position[] continues to find all matching positions.

```
In[10]:= Position[x, xi_ /; xi >= r, {1}, 1]
Out[10]= {{2}}
```

Positions are returned in extra list braces, which we remove here.

```
In[11]:= %[[1, 1]]
Out[11]= 2
```

■ 4.4.4.2 Folding to the Right

In Equation 4.4–1 we chose iteration along the first (left) argument of g. An equally valid choice is iteration along the second (right) argument:

$$x_i = g(y_{n-i+1}, x_{i-1}), \quad i = 1, \ldots, n. \tag{4.4–2}$$

This operation is not built in. Let us call it FoldRight[g, x_0, {y_1, ..., y_n}]. Implementation according to Equation 4.4–2 is straightforward if we reverse indices; see Listing 4.4–3.

The usual symbolic example shows the basic idea.

```
In[1]:= FoldRight[ g, x, {a, b, c, d} ]
Out[1]= g[a, g[b, g[c, g[d, x]]]]
```

For associative operations, such as addition, there is no difference between Fold and FoldRight.

```
In[2]:= FoldRight[ Plus, 0, {a, b, c, d} ]
Out[2]= a + b + c + d
```

The two operations are quite different for nonassociative operations, such as exponentiation. The first expression is

$$((1^a)^b)^c = 1^{abc} = 1\,,$$

while the second expression is

$$a^{b^{c^1}} = a^{b^c}\,.$$

```
In[3]:= Through[{FoldLeft, FoldRight}[Power, 1, {a, b, c}]]
                c
               b
Out[3]= {1, a  }
```

```
BeginPackage["ProgrammingInMathematica`FoldRight`"]

FoldRight::usage = "FoldRight[g, x, {y1, y2, ..., yn}] gives
    g[y1, g[y2, ..., g[yn, x]...]]."
FoldRightList::usage = "FoldRightList[g, x, {y1, y2, ..., yn}] gives
    {x, g[yn, x], ..., g[y2, ..., g[yn, x]...], g[y1, g[y2, ..., g[y1, x]...]]}."

FoldLeft = Fold          (* additional names for consistency *)
FoldLeftList = FoldList

Begin["`Private`"]

FoldRight[ g_, x0_, y_List ] :=
    Module[{x = x0},
        Do[ x = g[y[[i]], x], {i, Length[y], 1, -1} ];
        x
    ]

FoldRightList[ g_, x0_, y_List ] :=
    Module[{x = x0},
        Prepend[ Table[ x = g[y[[i]], x], {i, Length[y], 1, -1} ], x0 ]
    ]

End[ ]

Protect[ FoldRight, FoldRightList ]

EndPackage[ ]
```

Listing 4.4–3: FoldRight.m

4.5 Map and Apply

Not all programming languages give you the ability to treat functions like any other objects (symbols or numbers). In *Mathematica*, you can assign them to variables and they can be arguments to other functions. Two important commands that take functions as arguments are `Map[]` and `Apply[]`. We have already used them in a few places, but now we want to look at them in detail.

4.5.1 Mapping Functions onto Expressions

In Section 4.3.4 we have briefly encountered `Map[]`. `Map[`f`,` *list*`]` maps the function f over the elements of the list *list*. This means that it forms the expression f`[`e_i`]` for each element e_i of *list* and returns the list of the results. The second argument of `Map[]` need not be a list, however. Any expression of the form h`[`e_1`,` e_2`, ...,` e_n`]` will do. The result of the mapping is the expression h`[`f`[`e_1`],` f`[`e_2`], ...,` f`[`e_n`]]`.

The internal form of the expression is `Plus[a, b, c]`; therefore, we get `Plus[f[a], f[b], f[c]]`, which is printed as shown.

```
In[1]:= Map[ f, a + b + c ]
Out[1]= f[a] + f[b] + f[c]
```

The pure function used here extracts the second element of its argument. The result of the mapping is to extract the second element from each element of the list.

```
In[2]:= Map[ #[[2]]&, {a+b, 2 x y, z^2,
                      alpha (gamma + delta) / beta} ]

                    1
Out[2]= {b, x, 2, ----}
                  beta
```

`FullForm[]` is useful to understand what the second element is in each of the four elements of our list. Sometimes it is not what it appears to be (look at the fourth element).

```
In[3]:= FullForm[ {a+b, 2 x y, z^2,
                  alpha (gamma + delta) / beta} ]
Out[3]//FullForm=
 List[Plus[a, b], Times[2, x, y], Power[z, 2],
  Times[alpha, Power[beta, -1], Plus[delta, gamma]]]
```

The infix operator `/@` is often used for `Map[]`.

```
In[4]:= f /@ {a, b, c}
Out[4]= {f[a], f[b], f[c]}
```

`Map[`f`,` *expr*`,` *levelspec*`]` has an optional third argument that specifies the levels at which to map. The default level is `{1}`, that is, at the elements of *expr*. In a matrix, the entries are at level 2 (there are two levels of lists); if we want to map a function at these entries, we can use `Map[`f`,` *matrix*`, {2}]`. Note the difference between this and `Map[`f`,` *matrix*`, 2]`, which would map at all levels *up to* 2. `Map[`f`,` *expr*`, {-1}]` maps at all lowest levels, that is, at the atoms in *expr*. Level specifications are explained in Subsection 2.1.7 of the *Mathematica* book.

■ 4.5.1.1 Map at Particular Positions

`Map[]` always maps the function at all elements of the given levels. The command `MapAt[`*f*`,` *expr*`,` *poslist*`]` allows you to map a function at any given positions in your expression. (Positions and parts of expressions are treated in more detail in Subsection 2.1.4 of the *Mathematica* book.) *poslist* is a single position or a list of positions. The positions are lists of numbers that describe how to descend down the expression tree to that place.

This maps `f` at the first element of the second element (the symbol `c`).

```
In[5]:= MapAt[f, a b + c d + e f, {2, 1}]

Out[5]= a b + e f + d f[c]
```

This maps the square root function at position `{1}` (the term `a b`), at position `{2, 2}` (the symbol `c`) and at position `{3, 1}` (the number `4`). The usual evaluation rules then take effect to make the simplification `Sqrt[4]` ⟶ `2`.

```
In[6]:= MapAt[ Sqrt, a b + 2 c + 4/x,
               {{1}, {2, 2}, {3, 1}}]

                                     2
Out[6]= Sqrt[a b] + 2 Sqrt[c] + -
                                     x
```

Here is an expression.

```
In[8]:= expr = a + b/a + c E^(a+1)

              b      1 + a
Out[8]= a + - + c E
              a
```

If we wanted to map a function `f` at all occurrences of `a`, we first find all the positions of `a` in our expression.

```
In[9]:= Position[expr, a]

Out[9]= {{1}, {2, 1, 1}, {3, 2, 2, 2}}
```

This list of positions is in the right form for `MapAt` and can be used directly.

```
In[10]:= MapAt[f, expr, %]

             1 + f[a]      b
Out[10]= c E          + ---- + f[a]
                          f[a]
```

This particular example could have been done more easily with a replacement rule.

```
In[11]:= expr /. a -> f[a]

             1 + f[a]      b
Out[11]= c E          + ---- + f[a]
                          f[a]
```

Another example of the use of `Map[]` and `MapAt[]` will be given in Section 5.3.2.

■ 4.5.1.2 Position-Dependent Functions

When a function is mapped onto a list, as in `Map[`f`, {`e_1`, ..., `e_n`}]`, the resulting expressions f`[`e_1`]`, ..., f`[`e_n`]` are all evaluated in a uniform manner, independent of where they occurred in the original list. In some applications the function to perform on an element e_i may depend on its position i. The operation `MapIndexed[`g`, {`e_1`, `e_2`, ..., `e_n`}]` behaves essentially like `Map[]`, but it passes the position of each element as a second argument to f. The resulting expression is

`{`g`[`e_1`, {1}], `g`[`e_2`, {2}], ..., `g`[`e_n`, {`n`}]}.`

Therefore, the function g that is mapped must be a function of two arguments. The second argument is a position (note the list braces) that can be used to modify the operation performed on the first element.

This symbolic example should make it clear how MapIndexed[] works.

```
In[1]:= MapIndexed[ g, {a, b, c, d} ]
Out[1]= {g[a, {1}], g[b, {2}], g[c, {3}], g[d, {4}]}
```

Here we define a function of two arguments that raises the first argument to a power determined by the second argument. Note that the second argument is declared in a list pattern.

```
In[2]:= power[e_, {i_}] := e^i
```

The form of the arguments expected by power is exactly the form constructed by MapIndexed. Element e_i is raised to the i^{th} power.

```
In[3]:= MapIndexed[ power, {a, b, c, d} ]
                2   3   4
Out[3]= {a, b , c , d }
```

If we want to use a pure function, we cannot declare a list pattern for the second argument. Therefore, we extract the position from the list i using First[] in the body of the function. Here again, element e_i is raised to the i^{th} power.

```
In[4]:= MapIndexed[ Function[{e, i}, e^First[i]],
                    {1, 2, 3, 4, 5, 6, 7, 8} ]
Out[4]= {1, 4, 27, 256, 3125, 46656, 823543, 16777216}
```

The reason the positions are passed inside a list is that this form is extensible to mapping at levels other that the default first one.

Here we map the function g at level 2, that is, at the elements of the matrix. The positions are lists of length 2.

```
In[5]:= MapIndexed[ g, {{a, b}, {c, d}}, {2} ]
Out[5]= {{g[a, {1, 1}], g[b, {1, 2}]},
  {g[c, {2, 1}], g[d, {2, 2}]}}
```

■ 4.5.2 Apply

Apply[] implements a generalization of the usual notion of applying a function to an argument. When we speak of applying the function f to the expression *expr*, we mean to form the expression f[*expr*] and to evaluate it. If we want to apply a function to several arguments, things get more complicated. Assume you have a list l = {e_1, e_2, ..., e_n} of arguments and you want to compute f[e_1, e_2, ..., e_n]. Writing f[l] would be wrong; it would pass the whole expression {e_1, e_2, ..., e_n} as *one* argument to f in the form f[{e_1, e_2, ..., e_n}]. The expression Apply[f, l] does what you want. It forms the expression f[e_1, e_2, ..., e_n]. Looked at it from another point of view, it replaces the head of l by f. Remember that {e_1, e_2, ..., e_n} in internal form is just List[e_1, e_2, ..., e_n]. If we replace List by f we get f[e_1, e_2, ..., e_n]. If there is a special print form of an expression (as there is for lists or arithmetic operators), this replacement of the head may not look so obvious.

The head of this expression is Plus.

```
In[1]:= a + b + c
Out[1]= a + b + c
```

Replacing it by Times gives the product of the three terms in the sum above.

```
In[2]:= Apply[ Times, % ]
Out[2]= a b c
```

This simple definition finds the average of a list of numbers.

```
In[3]:= Average[ l_List ] := Apply[Plus, l] / Length[l]
```

Here is the expected value obtained from throwing a die.

```
In[4]:= Average[ {1, 2, 3, 4, 5, 6} ]

        7
Out[4]= -
        2
```

Of course, it also works for symbolic entries.

```
In[5]:= Average[ {a, b} ]

        a + b
Out[5]= -----
          2
```

`Apply[]` can be written with the infix operator `@@`.

```
In[6]:= f @@ {a, b, c}

Out[6]= f[a, b, c]
```

■ 4.6 Application: The Platonic Solids

The standard package Graphics/Polyhedra.m defines functions for generating lists of polygons representing the five Platonic solids (regular polyhedra) tetrahedron, cube, octahedron, dodecahedron, and icosahedron. It makes heavy use of `Map[]` and `Apply[]`.

■ 4.6.1 Generating the Polygons

The idea behind Polyhedra.m is to describe each solid that we want to render by the coordinates of all of its vertices and by data that describes which vertices belong to each of the faces. For each solid, we give rules for the two functions `Vertices[]` and `Faces[]`. The function `Vertices[`*name*`]` gives the list of vertex coordinates for the solid with name *name*. `Faces[`*name*`]` gives the list of faces. The function `Polyhedron[`*name*`]` uses the information from `Vertices[]` and `Faces[]` to assemble a list of polygons in a `Graphics3D[]` object. Listing 4.6–1 shows the relevant part of Graphics/Polyhedra.m.

How does `Polyhedron[]` work? It calls the auxiliary function `PolyGraphics3D[]` with the list of vertices and faces, respectively, of the solid we want. The code uses nested instances of `Map[]` written either in infix form as *f* `/@` *expr* or in prefix form as `Map[`*f*`,` *expr*`,` *level*`]`. As usual, we can unwind the nested function applications to see what it does in detail.

The variable `vertices` is set to the list of vertex coordinates of the tetrahedron.

```
In[1]:= vertices = Vertices[Tetrahedron]
Out[1]= {{0, 0, 1.73205}, {0, 1.63299, -0.57735},
  {-1.41421, -0.816497, -0.57735},
  {1.41421, -0.816497, -0.57735}}
```

The tetrahedron has four faces. Each one is a triangle and so the list of faces is a list of four triples, each one specifying which three vertices comprise that face.

```
In[2]:= faces = Faces[Tetrahedron]
Out[2]= {{1, 2, 3}, {1, 3, 4}, {1, 4, 2}, {2, 4, 3}}
```

We get the polygons by replacing the vertex numbers in the list of faces by the coordinates of the corresponding vertex. This statement gives us a list of faces, each of which is a list of the coordinates of the vertices.

```
In[3]:= Short[ (vertices[[#]]&) /@ faces, 3 ]
Out[3]//Short=
 {{{0, 0, 1.73205}, {0, 1.63299, -0.57735},
   {-1.41421, -0.816497, -0.57735}}, <<2>>,
  {{0, 1.63299, -0.57735}, <<2>>}}
```

This statement applies the optional scaling and translation of the solid. The defaults do not change the coordinates (we multiply each coordinate by 1.0 and add 0.0).

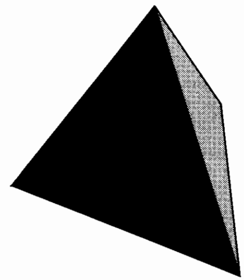
```
In[4]:= Map[ 1.0 # + {0.0, 0.0, 0.0} &, %, {2} ];
```

Now we wrap the function `Polygon[]` around each face of the tetrahedron and make it into a graphic object.

```
In[5]:= Graphics3D[ Polygon /@ % ]
Out[5]= -Graphics3D-
```

And finally we get the picture.

```
In[6]:= Show[ %, Boxed -> False ];
```



```
BeginPackage["Graphics`Polyhedra`"]

Polyhedron::usage = "Polyhedron[name] gives a Graphics3D object representing
    the specified solid centered at the origin and with unit distance to
    the midpoints of the edges.  Polyhedron[name, center, size] uses the
    given center and size. The possible names are in the list Polyhedra."

Vertices::usage = "Vertices[name] gives a list of the vertex coordinates
    for the named solid."

Faces::usage = "Faces[name] gives a list of the faces for the named solid. Each
    face is a list of the numbers of the vertices that comprise that face."

Polyhedra = {Tetrahedron, Cube, Octahedron, Dodecahedron, Icosahedron, Hexahedron,
    GreatDodecahedron, SmallStellatedDodecahedron,
    GreatStellatedDodecahedron, GreatIcosahedron}

Map[(Evaluate[#]::"usage" = ToString[#] <>
        " is a polyhedron, for use with the Polyhedron function.")&,
    Polyhedra]
:

Begin["`Private`"]

Polyhedron[name_Symbol, opts___ ] :=
    PolyGraphics3D[ Vertices[name], Faces[name], opts ] /; MemberQ[Polyhedra, name]

PolyGraphics3D[ vertices_, faces_, pos_:{0.0,0.0,0.0}, scale_:1.0 ] :=
    Graphics3D[ Polygon /@ Map[scale # + pos &, (vertices[[#]]&) /@ faces, {2}] ]

Tetrahedron /: Faces[Tetrahedron] =
     {{1, 2, 3}, {1, 3, 4}, {1, 4, 2}, {2, 4, 3}}

Tetrahedron /: Vertices[Tetrahedron] = N[
     {{0, 0, 3^(1/2)}, {0, (2*2^(1/2)*3^(1/2))/3, -3^(1/2)/3},
      {-2^(1/2), -(2^(1/2)*3^(1/2))/3, -3^(1/2)/3},
      {2^(1/2), -(2^(1/2)*3^(1/2))/3, -3^(1/2)/3}} ]
:

End[ ]

EndPackage[ ]
```

Listing 4.6–1: Part of Graphics/Polyhedra.m

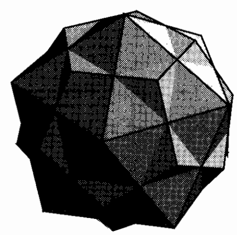
Input line 4 above deserves some more explanation. The variable *faces* is a list of lists of vertex numbers. The first of these lists is `{1, 2, 3}` specifying that the first face of our tetrahedron consists of vertices 1, 2, and 3. We then apply the pure function `vertices[[#]]&` to each of the lists of faces. The first of the entries will therefore be `vertices[[{1, 2, 3}]]` (before evaluation). If the argument of a part extraction (*expr*`[[...]]`) is not a single number, but a list of numbers, then it returns the *subexpression* consisting of the elements specified. Because `vertices` is a list, the subexpression will again be a list, consisting of the coordinates of the vertices number 1, 2, and 3.

This generates a list of two polyhedra to be shown together in one picture.

```
In[7]:= {Polyhedron[Icosahedron],
         Polyhedron[Dodecahedron]}
Out[7]= {-Graphics3D-, -Graphics3D-}
```

The vertex coordinates were chosen so that the edges of two dual solids intersect in the middle. The dodecahedron and icosahedron are duals, as are the cube and octahedron. The tetrahedron is dual to itself.

```
In[8]:= Show[ %, Boxed -> False ];
```



Note how the usage messages are attached to the symbols denoting the polyhedra, such as `Tetrahedron`. The function

`(Evaluate[#]::usage = "ToString[#] <> " text")&`

that assigns a usage message to its argument is mapped to the list `Polyhedra`.

■ 4.6.2 Manipulating Given Solids: Stellation

Once we have defined a solid in terms of its vertices and the list of faces, we can do more than just make pretty pictures. *Stellating* (or faceting) is one method of obtaining new solids from given ones. Each face is replaced by a pyramid with that face as its base and a new vertex, the apex of the pyramid, above the center of the face. The functional programming style makes it easy to replace each polygon in a graphics object by a pyramid.

We use a rule that replaces each expression of the form `Polygon[`*list*`]` by a list of polygons, one for each face of the pyramid. Here is the code from Polyhedra.m for the command `Stellate[`*graphics*`, `*ratio*`]` that applies this transformation to each polygon found in *graphics*.

```
BeginPackage["Polyhedra`"]
⋮
Stellate::usage = "Stellate[-Graphics3D-, (stellation ratio:2)] replaces all
    polygons in the graphics by a pyramid. Stellation ratios smaller
    than 1 give concave figures."

Begin["`Private`"]
⋮
StellateFace[face_, k_] :=
    Block[ { i, apex, n = Length[face] } ,
        apex = N[ k Apply[Plus, face] / n ] ;
        Table[ Polygon[{apex, face[[i]], face[[ Mod[i, n] + 1 ]] }], {i, n} ]
    ]

Stellate[ graphics_, k_:2.0 ] :=
    graphics /. Polygon[x_] :> StellateFace[x, k] /; NumberQ[N[k]]
⋮
End[ ]

EndPackage[ ]
```

Faceting polygons

The externally visible command is `Stellate`. It consists of a single rule that replaces each polygon by the result of applying `StellateFace[]` to its argument. `StellateFace[]` takes as argument a list of vertices and first computes their center. If `face` is the list {v_1, v_2, ..., v_n} in which each v_i is a list of three numbers (the coordinates of that vertex), then `Apply[Plus, face]` replaces the outer list by `Plus` and we get v_1+v_2+...+v_n. Because `Plus` is listable and each of the v_i is a list of three numbers, they are added component by component and we get a single list of the coordinates as a result (listability was explained in Section 4.3.5). This list is then divided component by component by `n`, the number of vertices of that face. The result is a point that lies in the center of the original face. We have done nothing other than taken the average of the vertices. The apex of the pyramid is this point stretched by an arbitrary factor `k`.

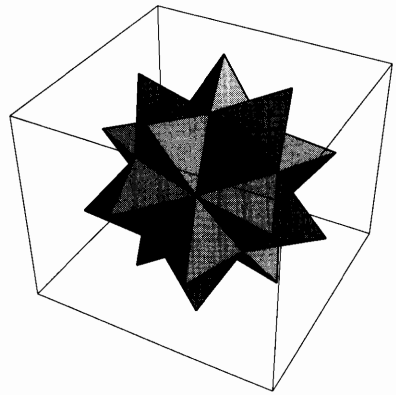
The faces that make up this pyramid are triangles with two vertices along the base and the third one being the apex. The `Table[]` command picks out all sets of two consecutive vertices of the original face and combines them with the apex to a polygon. The `Mod[]` function used to pick out the second of the vertices causes the last point to be 1 and not `n+1`, which would lie beyond the end of the list. (When i runs from 1 to n, then $\mathrm{mod}(i, n) + 1$ runs from 2 to n and then to 1.)

This gives a graphics object for the icosahedron.

```
In[1]:= Polyhedron[ Icosahedron ]
Out[1]= -Graphics3D-
```

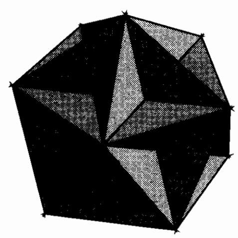
This figure should look familiar. It was at one point the logo of Wolfram Research, Inc.

```
In[2]:= Show[ Stellate[%] ];
```



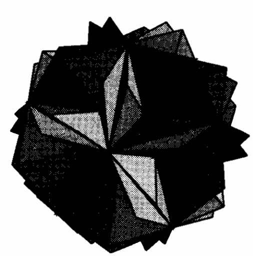
A stellation ratio smaller than 1 gives pyramids that point inward. The particular choice of $1/\sqrt{2}$ makes the new faces appear to be parts of pentagons with self-intersections. This solid is called a *great dodecahedron*.

```
In[3]:= Show[ Stellate[ %%, 1/Sqrt[2.0] ],
              Boxed -> False ];
```



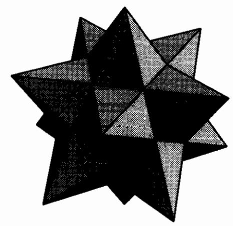
`Stellate[]` works on any set of polygon. We can iterate it, for example, giving this doubly stellated icosahedron.

```
In[4]:= Show[ Stellate[%, 1.5] ];
```



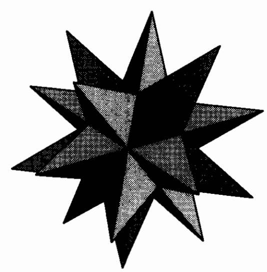
This is a *small stellated dodecahedron.* The triangular faces are part of 12 pentagrams.

```
In[5]:= Show[ Stellate[ Polyhedron[Dodecahedron],
                        Sqrt[5.0] ], Boxed -> False ];
```



A different stellation ratio turns our logo into a *great stellated dodecahedron.* It, too, consists of 12 pentagrams.

```
In[6]:= Show[ Stellate[Polyhedron[Icosahedron], 3],
              Boxed -> False ];
```



4.7 Operations on Lists and Matrices

Another important class of functional constructs manipulates the structure of expressions, mostly lists. `Transpose[]` and `Thread[]` rearrange the levels of nested lists. `Dot[]`, `Inner[]`, and `Outer[]` combine elements of two nested expressions in various ways. We shall look at all of these in turn.

4.7.1 Transposition and Threading

The operation of transposing a matrix interchanges rows and columns of a matrix. Besides its obvious application as the transpose function m^T of a matrix m, it can be used on multidimensional collections of data that are not properly matrices in the mathematical sense.

Here is the list of integers from 1 to 12.

```
In[1]:= n = Range[1, 12]
Out[1]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

This finds the time it takes to expand the expression $(a+b+c+d+1)^i$ for $i = 1, 2, \ldots, 12$.

```
In[2]:= Timing[ Expand[(a+b+c+d+1)^#] ][[1]]& /@ n
Out[2]= {0. Second, 0. Second, 0.02 Second, 0.09 Second,
  0.13 Second, 0.22 Second, 0.43 Second, 0.67 Second,
  0.98 Second, 1.34 Second, 1.82 Second, 2.48 Second}
```

This combines the values of n and the timing information in a list.

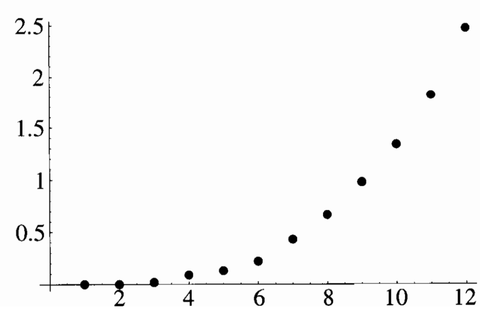
```
In[3]:= { n, % /. Second -> 1 }
Out[3]= {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12},
  {0., 0., 0.02, 0.09, 0.13, 0.22, 0.43, 0.67, 0.98,
   1.34, 1.82, 2.48}}
```

Transposing this list builds a list of pairs `{`i`,` $time_i$`}`.

```
In[4]:= Transpose[ % ]
Out[4]= {{1, 0.}, {2, 0.}, {3, 0.02}, {4, 0.09},
  {5, 0.13}, {6, 0.22}, {7, 0.43}, {8, 0.67}, {9, 0.98},
  {10, 1.34}, {11, 1.82}, {12, 2.48}}
```

We need the data in this form to make a list plot of it.

```
In[5]:= ListPlot[ %, PlotStyle -> PointSize[0.02] ];
```



`Transpose[]` takes an optional second argument that specifies which levels to interchange. If the levels in this level specification are not distinct, `Transpose[]` picks out diagonal elements in a matrix. This is useful, for example, to compute the *trace* of a matrix (the trace is the sum of the diagonal elements).

This definition computes the trace of a matrix.

```
In[6]:= MatrixTrace[m_?MatrixQ] :=
          Plus @@ Transpose[m, {1, 1}]
```

Here is a 3 by 3 matrix.

```
In[7]:= {{a, b, c}, {d, e, f}, {g, h, i}} // MatrixForm
Out[7]//MatrixForm= a   b   c

                    d   e   f

                    g   h   i
```

This is its trace, the sum of its diagonal elements.

```
In[8]:= MatrixTrace[ % ]
Out[8]= a + e + i
```

The nested levels of an expression that is to be transposed need not be lists. Any head will do. The heads of the expressions must be the same at all levels, however. Another operation, `Thread[]`, is needed to interchange levels with different heads.

This expression cannot be transposed; the head of the outermost level is `f`, but the inner level has head `List`.

```
In[9]:= f[{a, b}, {c, d}]
Out[9]= f[{a, b}, {c, d}]
```

`Thread[]` performs the operation.

```
In[10]:= Thread[ % ]
Out[10]= {f[a, c], f[b, d]}
```

Threading over lists can be made automatic by giving a function the attribute `Listable`.

```
In[11]:= SetAttributes[ g, Listable ];
```

Threading is done automatically.

```
In[12]:= g[ {a, b}, {c, d} ]
Out[12]= {g[a, c], g[b, d]}
```

We shall use `Thread[`*varlist* `->` *valuelist*`]` in Section 7.4 to obtain a list of replacements of values for a list of variables. The outer head is `Rule` and the inner head is `List`.

The result of `Thread[`f`[{`a_1`, ..., `a_n`}, {`b_1`, ..., `b_n`}, ...]` is

$$\{f[a_1,\ b_1,\ \ldots],\ \ldots,\ f[a_n,\ b_n,\ \ldots]\},$$

but the argument of `Thread[]` is evaluated before the rearrangements are done. Another operation, `MapThread[]`, can be used if this evaluation causes problems. The function `MapThread[`f`, {{`a_1`, ..., `a_n`}, {`b_1`, ..., `b_n`}, ...}]` gives the same result, without first forming and evaluating f`[{`a_1`, ..., `a_n`}, {`b_1`, ..., `b_n`}, ...]`.

After evaluating `f[{a, b}, {x, y}]`, which causes no harm here, threading gives this result.

```
In[13]:= Thread[ f[{a, b}, {x, y}] ]
Out[13]= {f[a, x], f[b, y]}
```

This computation involving `MapThread[]` gives the same result without evaluating `f[{a, b}, {x, y}]`.

```
In[14]:= MapThread[ f, {{a, b}, {x, y}} ]
Out[14]= {f[a, x], f[b, y]}
```

■ 4.7.2 Inner Products

The *inner product* or *dot product* is the usual matrix multiplication. You cannot use the normal multiplication * for multiplying matrices. First, matrix multiplication is not commutative; second, the normal multiplication is listable and is performed elementwise and not according to the special formula for multiplying matrices. If you use matrices and vectors with symbolic entries, it is easy to see what happens when you apply various multiplication operators (See also Subsections 3.7.5 and 3.7.11 of the *Mathematica* book).

This generates a symbolic 3 × 3 matrix.

```
In[1]:= (m = Array[a, {3,3}]) // MatrixForm
Out[1]//MatrixForm= a[1, 1]   a[1, 2]   a[1, 3]
                    a[2, 1]   a[2, 2]   a[2, 3]
                    a[3, 1]   a[3, 2]   a[3, 3]
```

This is vector of length 3. We can use it as either column- or row-vector. An explicit distinction between the two is not necessary.

```
In[2]:= v = Array[b, 3]
Out[2]= {b[1], b[2], b[3]}
```

Here v is used a column vector. Left multiplication by a matrix gives another column vector as result.

```
In[3]:= m . v // TableForm
Out[3]//TableForm=
 a[1, 1] b[1] + a[1, 2] b[2] + a[1, 3] b[3]
 a[2, 1] b[1] + a[2, 2] b[2] + a[2, 3] b[3]
 a[3, 1] b[1] + a[3, 2] b[2] + a[3, 3] b[3]
```

Here v is used as a row vector.

```
In[4]:= v . m // TableForm
Out[4]//TableForm=
 a[1, 1] b[1] + a[2, 1] b[2] + a[3, 1] b[3]
 a[1, 2] b[1] + a[2, 2] b[2] + a[3, 2] b[3]
 a[1, 3] b[1] + a[2, 3] b[2] + a[3, 3] b[3]
```

In contrast, this is what happens if we use the normal multiplication between a matrix and a vector.

```
In[5]:= m v // MatrixForm
Out[5]//MatrixForm=
 a[1, 1] b[1]   a[1, 2] b[1]   a[1, 3] b[1]
 a[2, 1] b[2]   a[2, 2] b[2]   a[2, 3] b[2]
 a[3, 1] b[3]   a[3, 2] b[3]   a[3, 3] b[3]
```

The dot product uses two operations to combine elements of its two arguments. It multiplies elements together and then adds up the resulting products. You can substitute your own functions for these two by using `Inner[`*multop*`,` m_1`,` m_2`,` *addop*`]`, where m_1 and m_2 are the two matrices or vectors whose inner product is to be formed, *multop* is the function to use for the multiplication, and *addop* is used for addition.

For an example, here is a very short definition for the divergence operator in Cartesian coordinates. Given an n-dimensional vector field $v = (e_1, e_2, \ldots, e_n)$ depending on n variables $(x_1, x_2, \ldots, x_n)$, then its divergence is given by the formula

$$\operatorname{div} v = \frac{\partial e_1}{\partial x_1} + \frac{\partial e_2}{\partial x_2} + \cdots + \frac{\partial e_n}{\partial x_n} .$$

We can recognize this as a generalized inner product with differentiation replacing multiplication.

In a similar manner, we can implement the gradient function using application of the derivative operator to a list of variables. Given a scalar field $s(x_1, x_2, \ldots, x_n)$, then its gradient is the vector field

$$\operatorname{grad} s = (\frac{\partial s}{\partial x_1}, \frac{\partial s}{\partial x_2}, \ldots, \frac{\partial s}{\partial x_n}).$$

Finally, the Laplacian of s can then be computed as $\nabla^2 s = \operatorname{div} \operatorname{grad} s$.

```
Div::usage = "Div[v, varlist] computes the divergence of
    the vectorfield v w.r.t. the given variables in Cartesian coordinates."

Grad::usage = "Grad[s, varlist] computes the gradient of s
    w.r.t. the given variables in Cartesian coordinates."

Laplacian::usage = "Laplacian[s, varlist] computes the Laplacian of
    the scalar field s w.r.t. the given variables in Cartesian coordinates."

Div[v_List, var_List] := Inner[ D, v, var, Plus ]

Grad[s_, var_List] := D[s, #]& /@ var

Laplacian[s_, var_List] := Div[ Grad[s, var], var ]
```

Part of VectorCalculus.m

Here is an example of its use with purely symbolic entries.

```
In[1]:= Laplacian[ v[x, y], {x, y} ]

         (0,2)           (2,0)
Out[1]= v     [x, y] + v     [x, y]
```

Here is a standard gravitational or electrical force field.

```
In[2]:= e = {x/(x^2 + y^2 + z^2)^(3/2),
             y/(x^2 + y^2 + z^2)^(3/2),
             z/(x^2 + y^2 + z^2)^(3/2)};
```

This is its divergence.

```
In[3]:= Div[e, {x, y, z}]

                   2                   2
               -3 x                 3 y
Out[3]= ---------------- - ---------------- -
          2    2    2 5/2    2    2    2 5/2
        (x  + y  + z )     (x  + y  + z )

               2
            3 z                   3
       ---------------- + ----------------
         2    2    2 5/2    2    2    2 3/2
       (x  + y  + z )     (x  + y  + z )
```

It needs some simplification to see that the result is correct.

```
In[4]:= Together[%]

Out[4]= 0
```

■ 4.7.3 Outer Products

The *outer product* is also described in Subsection 3.7.11 of the *Mathematica* book. For an application let us look at the Jacobian matrix. Given a list of expressions $(e_1, e_2, \ldots, e_n)$

that describe functions of the variables $(x_1, x_2, \ldots, x_m)$, then the Jacobian is an $n \times m$ matrix of partial derivatives

$$\begin{pmatrix} \frac{\partial e_1}{\partial x_1} & \frac{\partial e_1}{\partial x_2} & \cdots & \frac{\partial e_1}{\partial x_m} \\ \frac{\partial e_2}{\partial x_1} & \frac{\partial e_2}{\partial x_2} & \cdots & \frac{\partial e_2}{\partial x_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_n}{\partial x_1} & \frac{\partial e_n}{\partial x_2} & \cdots & \frac{\partial e_n}{\partial x_m} \end{pmatrix} .$$

This is an outer product with differentiation replacing multiplication. Here is the definition:

```
JacobianMatrix::usage = "JacobianMatrix[flist, varlist] computes the Jacobian of
    the functions flist w.r.t. the given variables."

JacobianMatrix[f_List, var_List] := Outer[ D, f, var ]
```

Definition of the Jacobian matrix using outer products

Here is a symbolic Jacobian.

```
In[5]:= JacobianMatrix[{f[x, y, z], g[x, y, z]},
                       {x, y, z}] //  MatrixForm

Out[5]//MatrixForm=

 (1,0,0)            (0,1,0)            (0,0,1)
f        [x, y, z]  f        [x, y, z]  f        [x, y, z]

 (1,0,0)            (0,1,0)            (0,0,1)
g        [x, y, z]  g        [x, y, z]  g        [x, y, z]
```

■ 4.7.4 Distribution

The distributive law of addition and multiplication states that

$$(a_1 + a_2)(b_1 + b_2) = a_1 b_1 + a_1 b_2 + a_2 b_1 + a_2 b_2 \,. \tag{4.7–1}$$

This operation is performed by `Distribute[(`a_1 `+` a_2`)(`b_1 `+` b_2`)]`. The operation can be generalized to other functions, besides addition and multiplication. It is a combinatorial operation that forms combinations of all elements appearing in the arguments of the inner function. More precisely,

`Distribute[`f`[`g`[`a_1`,` a_2`, ...],` g`[`b_1`,` b_2`, ...], ...],` g`,` f`]`

forms the expression

g`[`f`[`a_1`,` b_1`, ...], ...,` f`[`a_1`,` b_2`, ...], ...,` f`[`a_2`,` b_1`, ...], ...]`

where the arguments of f are all combinations of one argument from each g in the original expression.

Here is an example of the ordinary distributive law.

```
In[6]:= Distribute[ (a1+a2)(b1+b2+b3) ]

Out[6]= a1 b1 + a2 b1 + a1 b2 + a2 b2 + a1 b3 + a2 b3
```

Here is an example with symbolic `f` and `g`.

```
In[7]:= Distribute[f[g[a1, a2], g[b], g[c1, c2, c3]], g, f]
Out[7]= g[f[a1, b, c1], f[a1, b, c2], f[a1, b, c3],
  f[a2, b, c1], f[a2, b, c2], f[a2, b, c3]]
```

The default for g is `Plus`, and the default for f is the head of the expression, `h` in this case.

```
In[8]:= Distribute[ h[a1 + a2, b, c1 + c2 + c3] ]
Out[8]= h[a1, b, c1] + h[a1, b, c2] + h[a1, b, c3] +
  h[a2, b, c1] + h[a2, b, c2] + h[a2, b, c3]
```

In this example both f and g are `List`. The number of elements in the result is equal to the product of the lengths of the inner lists, here $2 \cdot 1 \cdot 3 = 6$.

```
In[9]:= Distribute[{{a1, a2}, b, {c1, c2, c3}}, List, List]
Out[9]= {{a1, b, c1}, {a1, b, c2}, {a1, b, c3},
  {a2, b, c1}, {a2, b, c2}, {a2, b, c3}}
```

The result of `Distribute[`f`[`g`[...], `g`[...], ...], `g`, `f`]` has head g and all elements have head f. Sometimes we want to replace these two heads by new ones, gp and fp, say. We could, of course, apply the substitution `{`f `->` fp`, `g `->` gp`}` to the result of `Distribute[]`, but this would often not work, because the intermediate result—before the substitution—is evaluated. Therefore, `Distribute[]` takes two additional optional arguments that specify the new heads to use in the result. The result of

`Distribute[`f`[`g`[...], `g`[...], ...], `g`, `f`, `gp`, `fp`]`

is

gp`[`fp`[...], ..., `fp`[...]].`

■ 4.7.5 Application: The Swinnerton-Dyer Polynomials

In one application we had to compute the product

$$s_n(x) = \prod \left(x \pm \sqrt{p}_1 \pm \sqrt{p}_2 \cdots \pm \sqrt{p}_n\right) \qquad (4.7\text{–}2)$$

where the product ranges over all sign combinations and p_i denotes the i^{th} prime number. Because for each term there are two choices of the sign, there are a total of 2^n terms. The terms and the product can be generated as an outer product and a distribution.

Here is a list of the square roots of the first three primes.

```
In[10]:= Sqrt[Prime[Range[3]]]
Out[10]= {Sqrt[2], Sqrt[3], Sqrt[5]}
```

Here we generate the terms $\pm\sqrt{p}_i$ for $i = 1, 2, 3$ in lists of length two.

```
In[11]:= Outer[ Times, %, {-1, 1} ]
Out[11]= {{-Sqrt[2], Sqrt[2]}, {-Sqrt[3], Sqrt[3]},
  {-Sqrt[5], Sqrt[5]}}
```

We append the variable `x` to the list of square roots.

```
In[12]:= Append[ %, x ]
Out[12]= {{-Sqrt[2], Sqrt[2]}, {-Sqrt[3], Sqrt[3]},
  {-Sqrt[5], Sqrt[5]}, x}
```

The terms in the product consist of one element from the inner lists each (and the variable `x`). We can generate them by distributing the inner lists over the outer ones. The new inner operation should be addition and the new outer operation should be multiplication.

```
In[13]:= Distribute[ %, List, List, Times, Plus ]
Out[13]= (-Sqrt[2] - Sqrt[3] - Sqrt[5] + x)
  (Sqrt[2] - Sqrt[3] - Sqrt[5] + x)
  (-Sqrt[2] + Sqrt[3] - Sqrt[5] + x)
  (Sqrt[2] + Sqrt[3] - Sqrt[5] + x)
  (-Sqrt[2] - Sqrt[3] + Sqrt[5] + x)
  (Sqrt[2] - Sqrt[3] + Sqrt[5] + x)
  (-Sqrt[2] + Sqrt[3] + Sqrt[5] + x)
  (Sqrt[2] + Sqrt[3] + Sqrt[5] + x)
```

Expanding out the terms leaves us with a polynomial with integer coefficients, the third *Swinnerton-Dyer polynomial.*

```
In[14]:= Expand[ % ]
                     2         4        6    8
Out[14]= 576 - 960 x  + 352 x  - 40 x  + x
```

Listing 4.7–1 shows a short program for generating the Swinnerton-Dyer polynomials for any n.

```
SwinnertonDyerP::usage = "SwinnertonDyerP[n, var] gives the minimal polynomial
    of the sum of the square roots of the first n primes."

Begin["`Private`"]

SwinnertonDyerP[ n_Integer?NonNegative, x_ ] :=
    Module[{arglist, poly},
        arglist = Outer[ Times, Sqrt[Prime[Range[n]]], {-1, 1} ];
        arglist = Append[ arglist, x ];
        poly = Distribute[ arglist, List, List, Times, Plus ];
        Expand[ poly ]
    ]

End[ ];
```

Listing 4.7–1: SwinnertonDyer1.m: Generating the Swinnerton-Dyer polynomials

Here is the next one of these polynomials. It has degree 16.

```
In[15]:= SwinnertonDyerP[4, z]
                             2              4             6
Out[15]= 46225 - 5596840 z  + 13950764 z  - 7453176 z  +

           8            10          12        14    16
  1513334 z  - 141912 z   + 6476 z   - 136 z   + z
```

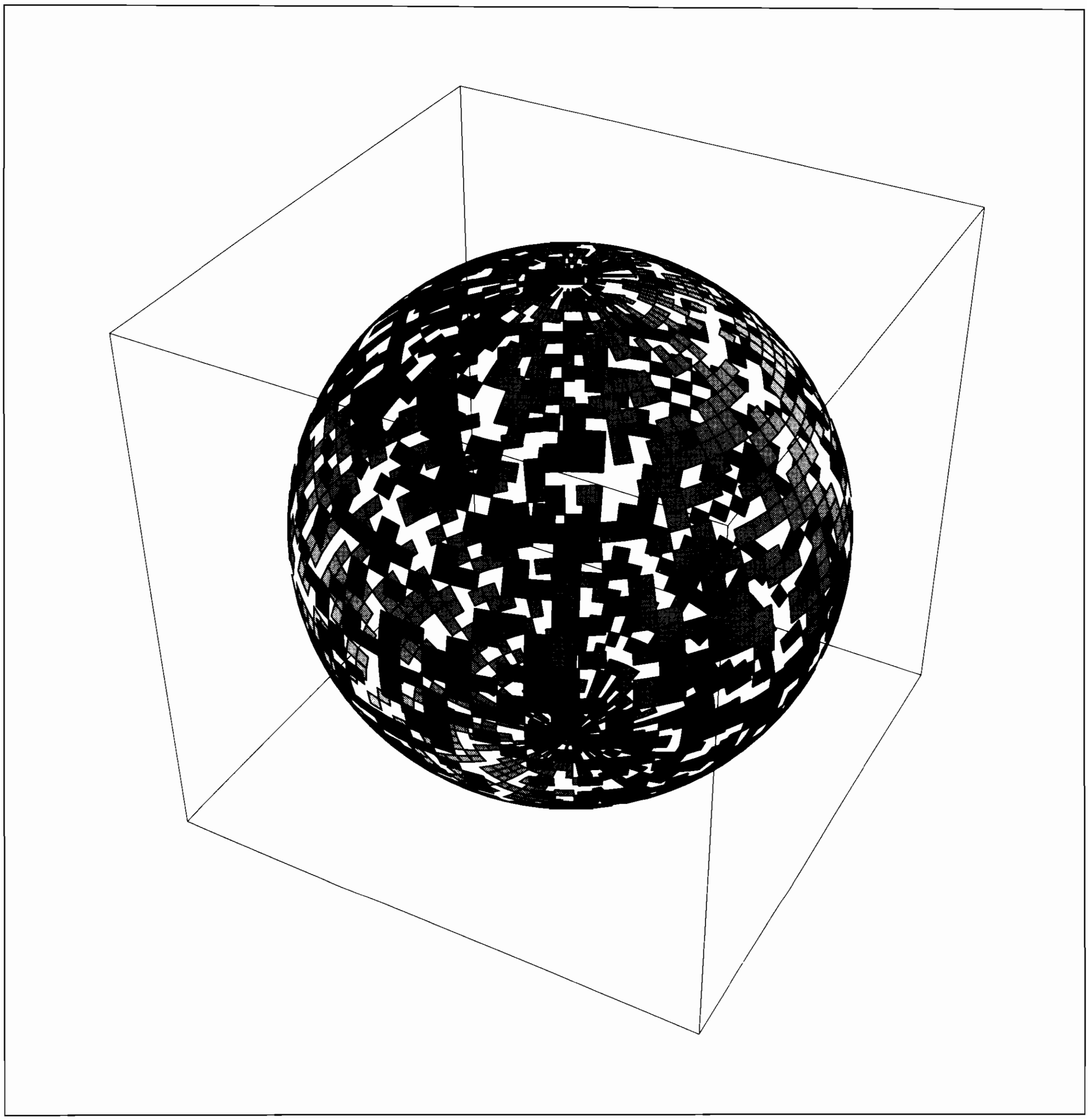
The Swinnerton-Dyer polynomials are the minimal polynomials of $\sum_{i=1}^{n} \sqrt{p_i}$. We can easily verify that the sum of the square roots of the first four primes is indeed a zero of this polynomial.

```
In[16]:= Expand[ % /. z -> Sqrt[2]+Sqrt[3]+Sqrt[5]+Sqrt[7] ]
Out[16]= 0
```

Expanding the product of 2^n factors, each one having $n+1$ terms, leads to huge intermediate expressions, before massive cancellation of all square roots occurs. We shall discuss a more efficient way to generate these polynomials in Section 5.5.4.2.

Chapter 5

Evaluation



In this chapter we look at various aspects of *Mathematica*'s way of evaluating expressions. The usual way of evaluating expressions is quite straightforward, but there are many exceptions and subtle points. Some knowledge of how expressions are evaluated is necessary to write good programs.

Section 1 looks at how rules are applied. There is an important difference between rules in *Mathematica* and procedures in traditional languages. It concerns the way pattern names are used inside the body of a rule. Clever use of this setup allows you to write functions that return other functions as results. This section shows you how to do this.

Pure functions are the topic of Section 2. A concept not available in most other languages, they are a very useful tool. We use them heavily in this book.

There are cases when you do not want to evaluate an argument of a function that is normally evaluated. On the other hand, there are a few built-in functions that do not evaluate their arguments. You can force evaluation in this case and prevent it in the former. Section 3 describes how to achieve this and what it is good for.

In Section 4 we look at ways to modify the normal sequential flow of control, how to exit from loops and procedures, and how to deal with error conditions.

Assigning values to variables is a simple concept, but (no surprise) in *Mathematica* this idea has advanced uses, treated in Section 5. A discussion of local definitions (within other definitions) follows. Finally, this section treats yet another subtlety encountered when defining rules. It has to do with the way the left side of a definition is evaluated. If you try to make a definition and you get strange error messages or the rule does not match the intended cases, try looking in this section for a possible cause of the problem.

Section 6 is about scoping of variables and symbols. This is an advanced topic; its understanding is not necessary for writing ordinary procedural programs.

About the illustration overleaf:

The `Sphere[]` command from the package Shapes.m creates polygons that approximate a sphere. In this picture we have randomly removed half of these polygons.

```
<< Graphics`Shapes`
Show[ Graphics3D[Select[Sphere[1, 72, 54], Random[]>0.5&]] ]
```

5.1 Evaluation of the Body of a Rule

There is an important difference between definitions in *Mathematica* and procedures in traditional languages. It concerns the way pattern names are used inside the body of a definition. This section discusses the differences and shows you how to emulate traditional parameter-passing mechanisms and how to take advantage of *Mathematica*'s substitution mechanism.

5.1.1 Pattern Names and Local Variables

In many programming languages formal parameters of functions are treated like local variables inside the body of the function (see Section 4.1). Because function definitions are simply transformation rules, things are different in *Mathematica*. Let us look at the issues involved in some more detail. Assume you give a rule like

`rule = f[x_] :> `*body*

or a definition like

`g[x_] := `*body*

and then evaluate `f[`*expr*`] /. rule` or `g[`*expr*`]`, respectively. The expressions `f[`*expr*`]` and `g[`*expr*`]` match the patterns `f[x_]` and `g[x_]`, respectively, with *expr* matching `x`. Evaluation then proceeds with the right side of the rule, the expression *body*. Before it is evaluated, all occurrences of `x` are replaced by *expr*, much in the same way as if you had given the expression

body `/. x ->` *expr*

except that *body* is not evaluated before the substitution (see Section 5.2.4 for more explanations on this topic). A consequence of this substitution is that names of patterns (such as `x` here) cannot be used as local variables. It is not possible to assign to them.

This definition tries to compute the sine of `x` and then return its square root.

```
In[1]:= f[x_] := (x = Sin[x]; Sqrt[x])
```

An error message results.

```
In[2]:= f[2.0]

Set::setraw: Cannot assign to raw object 2..

Out[2]= 1.41421
```

Simulating the evaluation of the body of the definition makes it clear what happened. After the substitution we would try to assign to the number `2.0`.

```
In[3]:= Hold[(x = Sin[x]; Sqrt[x])] /. x -> 2.0

Out[3]= Hold[2. = Sin[2.]; Sqrt[2.]]
```

If you want to use pattern names as local variables in the same way as procedure parameters are used in the programming languages **C** and Pascal, for example, you can use an initialized local variable.

```
f[x0_] :=
    Module[{x = x0},
        x = Sin[x];
        Sqrt[x]
    ]
```

Parameters as initialized local variables

This code corresponds to the following **C** function definition, assuming types `double` for the argument and result:

```
double
f(double x)
{
    x = sin(x);
    return sqrt(x);
}
```

Function parameters in **C**

Many procedural languages offer another parameter passing mechanism: parameter passing by reference (`var` parameters in PASCAL, pointers in **C**, references in C++). This style can be realized in *Mathematica* by suppressing the evaluation of the arguments through the attributes `HoldFirst`, `HoldRest`, or `HoldAll`. Here is a C++ function using a reference. It doubles the value of the referenced (global) variable.

```
void
twice(int &xref)
{
    xref = 2 * xref;
}
```

Reference parameters in C++

The equivalent *Mathematica* definition is this:

```
SetAttributes[twice, HoldAll]

twice[xref_Symbol] := (
    xref = 2 xref;
)
```

Reference parameters in *Mathematica*

Here is a global variable having the value 5.

```
In[4]:= a = 5;
```

Giving it as argument to the function `twice` should double its value.

```
In[5]:= twice[a]
```

Indeed, this is what happened.

```
In[6]:= a
Out[6]= 10
```

5.1.2 Local Constants

Local constants are less widely known than are local variables. Consider this expression:

$$x(y-1)(1-x^2)+(y-1)^2-y(1-x^2).$$

The subterms $y-1$ and $1-x^2$ occur in several places. This expression can be written in a simpler way by introducing two auxiliary variables $a = y-1$ and $b = 1-x^2$ and then writing

$$xab+a^2-yb.$$

The variables a and b should, of course, be local to this expression, just like local variables of a function, but we do not really need local *variables*, because we do not try to assign to them. We look only at their initial values. Within our expression, a and b are constants. The construct

With[{*const*$_1$ = *val*$_1$, ..., *const*$_n$ = *val*$_n$}, *body*]

implements this idea. Every occurrence of the declared constants is replaced by their values. As with pattern variables, it is not possible to assign to them. Their scope is *body*. Such a construct is usually available in functional languages. In LISP it is called `let`.

This is the translation of the formula given above into *Mathematica*. An added benefit is that the terms `y - 1` and `1 - x^2` are evaluated only once.

```
In[7]:= With[ {a = y - 1, b = 1 - x^2}, x a b + a^2 - y b ]

                2              2         2
Out[7]= x (1 - x ) (-1 + y) + (-1 + y)  - (1 - x ) y
```

`With[]` helps to write concise functional code.

```
In[8]:= ShiftedSum[x_, s_, n_] :=
          Module[{i},
             With[{y = x - s}, Sum[ y^i, {i, 0, n} ] ]
          ]
```

`Module[]` and `With[]` prevent also conflicts of names of symbols (here for `y` and `i`).

```
In[9]:= ShiftedSum[y, i, 4]

                              2          3          4
Out[9]= 1 - i + y + (-i + y)  + (-i + y)  + (-i + y)
```

Section 5.6 will present more explanations and examples of scoping constructs such as `With[]`.

5.1.3 Functions That Return Functions

The mechanism explained in the previous subsection is important if we want to write functions that return other functions. Let us write a function `MultByN[`n`]` that returns a pure function that multiplies its argument by n.

Here we use the fact that replacement of the argument value is done everywhere on the right side, even inside pure functions, whose body is not evaluated.

```
In[1]:= MultByN[n_] := Function[x, x n]
```

m5 is a function that multiplies its argument by 5. To guard against possible conflicts of names of symbols, the local variable x has been renamed.

```
In[2]:= m5 = MultByN[5]
Out[2]= Function[x$, x$ 5]
```

Applied to the argument 7 we get 35.

```
In[3]:= m5[7]
Out[3]= 35
```

A function MultByN2[n] that returns a function that multiplies its argument by n^2 is harder to write. We need to perform some computation (n^2 in this example) and then insert its result into the body of the resulting pure function. The construct With[{*var* = *val*}, *body*] can be used (see Section 5.1.2).

Here is our definition.

```
In[4]:= MultByN2[n_] :=
          With[{n2 = n^2},
            Function[x, x n2]
          ]
```

The body of With[] is evaluated with all occurrences of n2 replaced by its value, 5. This replacement happens also inside Function[], which is not evaluated further. To guard against possible conflicts of names of symbols the local variable x has been renamed.

```
In[5]:= MultByN2[5]
Out[5]= Function[x$, x$ 25]
```

Applying the returned function to 7 we get the expected result, because $7 \cdot 5^2 = 175$.

```
In[6]:= %[7]
Out[6]= 175
```

Here we see why the renaming of the variable is necessary. This function multiplies its argument by x^2, independent of the name of its local variable.

```
In[7]:= MultByN2[x]
                         2
Out[7]= Function[x$, x$ x ]
```

The function MultByN2 could be written without With, in the form

```
MultByN2[n_] := Function[x, x n^2].
```

Such a definition is inefficient, however, because the square is computed every time the resulting function is used, not just once when it is defined.

The operation of differentiation is also a higher-level operation. It takes a function as argument and returns its derivative, again a function.

Here is a sample function definition.

```
In[1]:= f[x_] := 1 + x^2
```

This computes the first derivative of f. Because we do not have a name for this function, it is returned as a pure function.

```
In[2]:= f'
Out[2]= 2 #1 &
```

Applying it to an argument gives the expected result.

```
In[3]:= %[z]
Out[3]= 2 z
```

The argument of the derivative operator can, of course, itself be a pure function. This is the same function as `f` above and so, too, is its derivative.

```
In[4]:= (1 + #^2)&'
Out[4]= 2 #1 &
```

The derivative of a pure function is computed by differentiating its body with respect to its variable, in this case computing `D[y^x, x]`. The result is, however, not simplified because the body of a pure function must not be evaluated.

```
In[5]:= Function[x, y^x]'
                       x
Out[5]= Function[x, y   1 Log[y]]
```

As soon as the function is applied to an argument, it is fully evaluated and simplified.

```
In[6]:= %[z]
         z
Out[6]= y  Log[y]
```

■ 5.1.4 Application: Programmed Definition of Functions

In Section 5.1.3 we have seen a way to define functions that return functions as their values. In this section, we want to look at functions that define rules for another function. For an example, assume we need to set up a variety of *step functions*. A step function $f(x)$ is a function that takes on just two values, one value a if the argument x is less than some value x_0 and another value b if the argument is greater than x_0. In *Mathematica*, this can easily be defined as follows:

```
f[x_] /; x <= x0  := a
f[x_] /; x >  x0  := b
```

Defining a step function

Now we want to write a function `StepFunction[`f`, `a`, `x_0`, `b`]` that sets up such a definition for f. This is easy:

```
StepFunction[f_Symbol, a_, x0_, b_] := (
    f[x_] /; x <= x0  := a;
    f[x_] /; x >  x0  := b;
)
```

The first version of `StepFunction[]`

This works because the values of the pattern variables `f`, `a`, `x0`, and `b` are *substituted* in the body of the rule. Evaluating the body globally defines the rules for `f`, as if we had typed them in. The name of the variable `x` used in the rules defined for `f` does not matter at all; there is no need to pass it as a parameter of `StepFunction`. Note that the parentheses are necessary.

This defines the *sign* function. (It is already built in as `Sign[]`.)

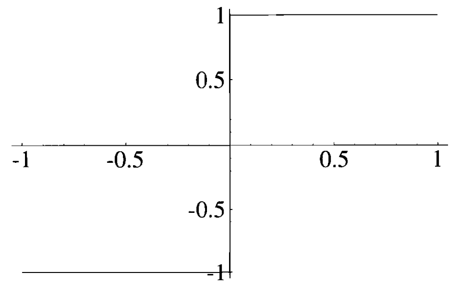
```
In[1]:= StepFunction[Signum, -1, 0, 1]
```

The rules have indeed been defined.

```
In[2]:= ?Signum
Global`Signum

Signum[ProgrammingInMathematica`MakeFunctions`Private`x
      _] /; ProgrammingInMathematica`MakeFunctions`Privat\
e`x <= 0 := -1

Signum[ProgrammingInMathematica`MakeFunctions`Private`x
      _] /; ProgrammingInMathematica`MakeFunctions`Privat\
e`x > 0 := 1
```

Here is a plot of it.

```
In[3]:= Plot[ Signum[x], {x, -1, 1} ];
```



In this example, the right side and the condition of the rules involved only the parameters of the function `StepFunction[]` itself. They were, therefore, substituted correctly everywhere inside the body of `StepFunction[]`. If the right side or the condition is more complicated and has to be computed first, this would not work. The definitions set up for `f` are not evaluated inside `StepFunction[]`. We have to use substitution to get the computed values inside these definitions in the same way that we did in Section 5.1.3. Another approach is to write an auxiliary function `MakeRuleConditional[]` that defines a rule. We can then pass the parts of the rule as parameters and again use substitution.

The package MakeFunctions.m (Listing 5.1–1) implements this auxiliary function and uses it to define the two functions `StepFunction[]` and `LinearFunction[]`.

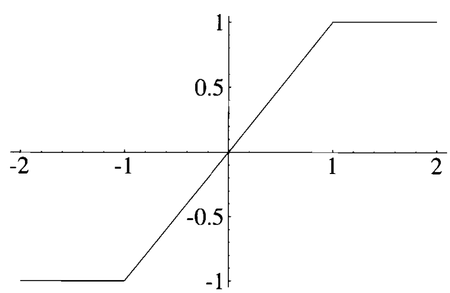
`LinearFunction[`f`, `a`, `x_0`, `x_1`, `b`]` defines $f(x)$ to be a function whose values are a for $x \leq x_0$, b for $x \geq x_1$, and linearly increase from a to b between x_0 and x_1.

This defines rules for `g`.

```
In[4]:= LinearFunction[g, -1, -1, 1, 1]
```

And here is a picture of `g`.

```
In[5]:= Plot[ g[y], {y, -2, 2} ];
```



The functions `MakeRule[]` and `MakeRuleConditional[]` do not evaluate their arguments. This is desirable so that unevaluated right sides and conditions can be specified. If these arguments contain values of local variables (such as `slope`), their values must be put inside the argument. This can be done most easily using `With[]`.

```
BeginPackage["ProgrammingInMathematica`MakeFunctions`"]

StepFunction::usage = "StepFunction[f, a, x0, b] defines rules for f
    such that f[x] = a for x <= x0, f[x] = b for x > x0."

LinearFunction::usage = "LinearFunction[f, a, x0, x1, b] defines rules for f
    such that f[x] = a for x <= x0, f[x] = b for x >= x1 and
    f increases linearly from a to b between x0 and x1."

MakeRule::usage = "MakeRule[f, x, rhs] globally defines the rule f[x_] := rhs."

MakeRuleConditional::usage = "MakeRuleConditional[f, x, rhs, cond]
    globally defines the rule f[x_] /; cond := rhs."

Begin["`Private`"]

SetAttributes[MakeRule, HoldAll]

MakeRule[f_Symbol, var_Symbol, rhs_] := f[var_] := rhs

SetAttributes[MakeRuleConditional, HoldAll]

MakeRuleConditional[f_Symbol, var_Symbol, rhs_, condition_] :=
    f[var_] /; condition := rhs

StepFunction[f_Symbol, a_, x0_, b_] := (
    MakeRuleConditional[f, x, a, x <= x0];
    MakeRuleConditional[f, x, b, x >  x0];
    )

LinearFunction[f_Symbol, a_, x0_, x1_, b_] :=
    With[{slope = (b-a)/(x1-x0)},
        MakeRuleConditional[f, x, a, x <= x0];
        MakeRuleConditional[f, x, a + (x-x0) slope, x0 < x < x1];
        MakeRuleConditional[f, x, b, x >= x1];
    ]

End[]

Protect[StepFunction, LinearFunction, MakeRule, MakeRuleConditional]

EndPackage[]
```

Listing 5.1–1: MakeFunctions.m: Utilities for defining functions

5.2 Pure Functions

We have already encountered *pure functions* several times. They are discussed in Subsection 2.2.5 of the *Mathematica* book. Given an expression *expr* involving some variable x, you can think of *expr* as describing a function with that variable being the argument. If you need to refer to that function by name, you can either use a symbol f by giving a rule for f in the form

f`[`x`_] :=` *expr*

or you can use the object

`Function[`x`, `*expr*`]`.

Either f or `Function[`x`, `*expr*`]` is a name for that function and you can use the two interchangeably. To apply them to an argument *arg*, you write f`[`*arg*`]` in the familiar way or `Function[`x`, `*expr*`][`*arg*`]`, the latter perhaps looking rather strange at first.

This defines `f` to be the function with $f(x) = 1 + x^2$.

```
In[1]:= f[x_] := 1 + x^2
```

`g` is set to be a pure function that describes the same function as `f`.

```
In[2]:= g = Function[x, 1 + x^2]

                          2
Out[2]= Function[x, 1 + x ]
```

They can be used in the same way.

```
In[3]:= {f[3], g[3]}

Out[3]= {10, 10}
```

There are cases where a named function, like `f` above, cannot be used. Examples are functions that return functions as their result, which were discussed in Section 5.1.3.

5.2.1 Short Forms of Pure Functions

The name of the formal parameter in a pure function does not matter. `Function[x, x^2]` is the same function as `Function[y, y^2]`. This fact is easy to see if you apply such a function to an argument: the result is the same in both cases.

Here is the first function applied to the argument `a`.

```
In[4]:= Function[x, x^2][a]

         2
Out[4]= a
```

And here is the second one, applied to the same argument `a`.

```
In[5]:= Function[y, y^2][a]

         2
Out[5]= a
```

Because the names of the variables in a pure function do not matter, *Mathematica* can provide special symbols for denoting these variables. The expressions `#1`, `#2`, ... are used for the first, second, ... variable. The internal form of `#`i is `Slot[`i`]`. If you use these forms, you do not give the first argument of `Function[]` that serves to declare the names of the variables, that is, instead of `Function[x, x^2]` you simply write `Function[#1^2]`. A convenient abbreviation for `#1` is `#`, so we can simplify our example a bit more: `Function[#^2]`.

Finally, there is a postfix operator `&` for `Function`. That is, *body*`&` is the same as `Function[`*body*`]`. Using this operator, we arrive at the shortest form our example can take: `#^2&`. We have used this form quite frequently in this book.

Here you can see the internal form of this short form of a pure function.

```
In[6]:= FullForm[#^2&]
Out[6]//FullForm= Function[Power[Slot[1], 2]]
```

The operator `&` has a very low priority, just above assignment. Therefore, x `=` *body*`&` is understood as x `= Function[`*body*`]`, but x `->` *body*`&` is interpreted as `Function[`x `->` *body*`]`. If you want the right side of the rule to be the pure function, use x `-> (`*body*`&)`. Beware of x `-> (`*body*`)&`. Another case where the low priority of pure functions requires the use parentheses is in predicates for patterns: x`_?`*body*`&` is not the same as x`_?(`*body*`&)`. The latter is usually correct.

The whole expression to the left of `&` is part of the body of the pure function. No parentheses are necessary.

```
In[7]:= 1 + # + #^2 & [5]
Out[7]= 31
```

Parentheses around the pure function are needed here; otherwise the whole rule would be considered part of the body of the function.

```
In[8]:= h[a] /. h -> (#^2 &)
         2
Out[8]= a
```

The use of `#`i introduces an ambiguity in the number of arguments the pure function requires.

The long form specifies at least two arguments; therefore, we get this error message if we provide only one argument.

```
In[9]:= Function[{a, b}, a][1]
Function::fpct:
   Too many parameters in {a, b} to be filled from
    Function[{a, b}, a][1].
Out[9]= Function[{a, b}, a][1]
```

The short form specifies only that there should be at least one argument.

```
In[10]:= Function[#1][1]
Out[10]= 1
```

Any extra arguments are ignored.

```
In[11]:= Function[#1][a, b, c]
Out[11]= a
```

■ 5.2.2 Constant Pure Functions

The variables need not occur at all in the body of a pure function. The expression `1&` is a constant function that—independent of the values of its arguments—always returns 1.

The constant pure function applied to any argument always returns the same value.

```
In[12]:= 1&[5]
Out[12]= 1
```

Because no variables occur in the body of the pure function, the minimum number of arguments is zero.

```
In[13]:= 1&[]
Out[13]= 1
```

In this form, however, one argument is required, even though it does not appear in the body of the function.

```
In[14]:= Function[x, 1][]
Function::fpct:
   Too many parameters in {x} to be filled from
    Function[x, 1][].
Out[14]= Function[x, 1][]
```

This defines a function `constant[`*val*`]` whose value is a constant pure function that always returns *val*.

```
In[15]:= constant[x_] := x&
```

This defines `k7` to be a function that always returns 7.

```
In[16]:= k7 = constant[7]
Out[16]= 7 &
```

Whatever the argument, it returns 7.

```
In[17]:= k7[666]
Out[17]= 7
```

■ 5.2.3 Attributes of Pure Functions

The evaluation of an expression *f*`[`*arguments*`]` can be modified by attributes of the head symbol *f*. These attributes are `Listable`, `HoldFirst`, `HoldRest`, `HoldAllComplete`, and `SequenceHold`. Only symbols can have attributes. To remedy this limitation, a pure function can take an optional third argument: a list of attributes to use when this function is applied to arguments.

This pure function returns the length of its unevaluated argument. Note how the argument is preserved in unevaluated form by wrapping it in `Unevaluated`.

```
In[18]:= Function[{x}, Length[Unevaluated[x]],
                  {HoldAll}][1+1+1]
Out[18]= 3
```

An ordinary function evaluates its argument. The result is the length of the number 3.

```
In[19]:= Function[{x}, Length[Unevaluated[x]]][1+1+1]
Out[19]= 0
```

Attributes can be given only for pure functions with an explicit parameter list in the form `Function[`*parameters*`,` *body*`,` *attributes*`]`. The short form `Function[`*body*`]` (with parameters of the form `#`*n*) does not take a list of attributes. An undocumented feature allows the list of parameters to be given as `Null` to indicate that anonymous parameters `#`*n* are implied.

This pure function is equivalent to `#1+#2&`.

```
In[20]:= Function[Null, #1+#2][a, b]
Out[20]= a + b
```

We can use this feature to specify attributes to such pure functions.

```
In[21]:= Function[Null, Length[Unevaluated[#]],
                  {HoldAll}][1+1+1]

Out[21]= 3
```

■ 5.2.4 Advanced Topic: Theoretical Properties

The study of formal properties of systems of pure functions is a branch of mathematics called *λ-calculus*. Three of the theorems of λ-calculus are of particular interest for *Mathematica*'s pure functions. In λ-calculus, they are called β reduction, α conversion, and η conversion.

The first theorem (β reduction) states how pure functions are applied to arguments. This is done by replacing every occurrence of the formal variable in the body of the pure function by the argument. In *Mathematica* notation, the expression

`Function[`x`, `*body*`][`*arg*`]`

is evaluated essentially as

`ReleaseHold[ Hold[`*body*`] /. HoldPattern[`x`] -> `*arg*`].`

This is the same as

body `/.` x `->` *arg*

except that *body* and x are not evaluated before the rule is applied. Note that this is similar to how the body of a rule is evaluated, as we have seen in Section 5.1.

We set `x` to 0 so that we would later detect any attempt to evaluate the body of a pure function involving `x` before the argument is substituted for `x`.

```
In[1]:= x = 0

Out[1]= 0
```

This gives what we expect and is not influenced by the global value for `x`.

```
In[2]:= Function[x, 1/x][4]

         1
Out[2]= -
         4
```

This expression is equivalent. It shows how pure functions are applied to arguments.

```
In[3]:= ReleaseHold[ Hold[1/x] /. HoldPattern[x] -> 4 ]

         1
Out[3]= -
         4
```

The second theorem (α conversion) states essentially that the name of the formal variable does not matter. If we have a pure function `Function[x, `*expr*`]`, replacing the variable `x` by `y` everywhere in *expr* describes the exact same function.

Here is a sample pure function.

```
In[1]:= Function[x, Sqrt[x] + Sin[x] + E^(2x)]

                                         2 x
Out[1]= Function[x, Sqrt[x] + Sin[x] + E    ]
```

This expression describes the same function.

```
In[2]:= % /. x -> y

                                         2 y
Out[2]= Function[y, Sqrt[y] + Sin[y] + E    ]
```

As we have seen, the expression `Function[x, body][arg]` is evaluated by substituting the value of the argument *arg* for every occurrence of the variable `x` in *body*. Obviously the name of the variable does not matter. This is only true, however, if the new name `y` did not occur in *body* before the replacement. *Mathematica* treats the name of the variable in a pure function as local. It renames the variable if necessary to avoid any clash with a symbol that is brought into the scope of the pure function. For a detailed explanation of what happens if there *are* conflicts of names, see Section 5.6.

This pure function adds `y` to its argument.

```
In[3]:= Function[x, x + y]
Out[3]= Function[x, x + y]
```

Substitution does not pay attention to scoping rules and violates the semantics of λ-calculus. The resulting function *doubles* its argument.

```
In[4]:= % /. x -> y
Out[4]= Function[y, y + y]
```

`With` does the right thing: it ignores formal parameters in its body and does not substitute them.

```
In[5]:= With[{x = y}, Function[x, x + y]]
Out[5]= Function[x, x + y]
```

In this case it renames the formal parameter to preserve the semantics. This function adds `x` to its argument.

```
In[6]:= With[{y = x}, Function[x, x + y]]
Out[6]= Function[x$, x$ + x]
```

The third theorem (η conversion) states that the pure function `Function[var, f[var]]` is the same function as simply f itself.

This is merely a complicated way of writing the sine function.

```
In[7]:= Function[x, Sin[x]]
Out[7]= Function[x, Sin[x]]
```

Applying it to an argument gives the same as just applying `Sin` itself to that argument.

```
In[8]:= %[z]
Out[8]= Sin[z]
```

The reverse conversion, which turns f into `Function[var, f[var], attribute]`, can be useful to give a function an attribute during just one particular application (see Section 5.2.3).

The (symbolic) function `f` is not listable.

```
In[9]:= f[ {a, {b, c}} ]
Out[9]= f[{a, {b, c}}]
```

Now it is for this one use.

```
In[10]:= Function[x, f[x], Listable][ {a, {b, c}} ]
Out[10]= {f[a], {f[b], f[c]}}
```

`Thread[]` is not good enough for this purpose, because it deals with only one level of lists.

```
In[11]:= Thread[ f[ {a, {b, c}} ] ]
Out[11]= {f[a], f[{b, c}]}
```

5.3 Nonstandard Evaluation

Mathematica follows a simple strategy when evaluating an expression such as *f*[*args*]: evaluate *f* and the arguments, then apply any rules that match. Sometimes this simple evaluation scheme is not adequate. This section shows how to modify it. There are also some built-in functions that do not evaluate their arguments. We show how we can force their evaluation in cases where this is necessary.

5.3.1 Functions That Do Not Evaluate Their Arguments

Normally, functions evaluate their arguments before any rules for that function take effect. There are, however, special cases. We have already seen such examples, namely all the iterators. The prototypical function that does not evaluate its argument is `Hold[`*expr*`]`. It does nothing, but its presence prevents *expr* from being evaluated. For more on evaluation, see Section 2.5 of the *Mathematica* book. If you set the attribute `HoldAll` for a function, it will not evaluate its arguments. If you give a rule such as `f[e_] :=` *body*, then the unevaluated argument is substituted for every occurrence of `e` in the body of the rule. Unless such an occurrence is again in a function that does not evaluate its arguments, it will be evaluated there, inside your function.

As an example, here is a function `PrintTime[]` that prints the time it takes to evaluate its argument and then returns the result of that evaluation (Listing 5.3–1). It is clear that it must not evaluate its argument before it passes it to the built-in function `Timing[]` that does the time measurement.

```
PrintTime::usage = "PrintTime[expr] prints the time it takes
    to evaluate expr and returns the result of the evaluation."

Begin["`Private`"]

SetAttributes[PrintTime, HoldAll]

PrintTime[expr_] :=
    With[{timing = Timing[expr]},
        Print[ timing[[1]] ];
        timing[[2]]
    ]

End[];
```

Listing 5.3–1: PrintTime.m: Printing evaluation timings

The unevaluated argument is passed to `Timing[]` inside `PrintTime[]`. `Timing[]` itself does not evaluate its argument, either. It is evaluated inside the built-in code of `Timing[]`. `Timing[]` returns a list {*time*, *result*}. We print the time and return the result.

The printing of the time is a side effect that does not interfere with the normal course of the computation.

```
In[1]:= PrintTime[ Factor[x^10 - y^10] ]
0.03 Second
                             4    3      2  2      3    4
Out[1]= (x - y) (x + y) (x  - x  y + x  y  - x y  + y )
  4    3      2  2      3    4
 (x  + x  y + x  y  + x y  + y )
```

Timing[] returns this list and so can disturb the normal flow of computation, whereas PrintTime[] is almost invisible. Timing information is not very reproducible, and so the two times might be different.

```
In[2]:= Timing[ Factor[x^10 - y^10] ]
Out[2]= {0.03 Second, (x - y) (x + y)
   4    3      2  2      3    4
  (x  - x  y + x  y  - x y  + y )
   4    3      2  2      3    4
  (x  + x  y + x  y  + x y  + y )}
```

We shall present a way of keeping track of the timing for each command automatically in Section 8.1.2.

The evaluation of arguments can be controlled further by the attributes HoldFirst and HoldRest. They prevent the evaluation of the first and remaining arguments, respectively. Together they constitute the attribute HoldAll.

Here we use two symbols as undefined functions to show the effects of HoldFirst and HoldRest.

```
In[3]:= SetAttributes[f1, HoldFirst];\
        SetAttributes[f2, HoldRest];
```

The attributes suppress the evaluation of the respective arguments.

```
In[4]:= { f1[1+1, 2+2, 3+3], f2[1+1, 2+2, 3+3] }
Out[4]= {f1[1 + 1, 4, 6], f2[2, 2 + 2, 3 + 3]}
```

Together, they are referred to as a single attribute, HoldAll.

```
In[5]:= SetAttributes[f, {HoldFirst, HoldRest}];\
        Attributes[f]
Out[5]= {HoldAll}
```

The attributes HoldFirst and HoldRest, discussed in the previous subsection, affect only the normal evaluation sequence. Other forms of evaluation, such as upvalues, splicing of sequences, or flat and orderless properties, are still applied. To keep an expression with head h in completely unevaluated, inert form, give h the attribute HoldAllComplete. The standard container with this attribute is HoldComplete (similar to Hold[], which is the standard container with the attribute HoldAll).

Here is the definition of an upvalue for the symbol up that triggers whenever it appears as the first argument of any symbol.

```
In[6]:= up/: _Symbol[up, ___] := gotcha
```

The attribute HoldAll of Hold[] does not prevent the evaluation of upvalues.

```
In[7]:= Hold[ up ]
Out[7]= gotcha
```

HoldComplete is a stronger form of Hold. It has the attribute HoldAllComplete.

```
In[8]:= Attributes[ HoldComplete ]
Out[8]= {HoldAllComplete, Protected}
```

The attribute HoldAllComplete prevents any form of evaluation of the arguments of HoldComplete[]. The upvalue is not triggered.

```
In[9]:= HoldComplete[ up ]
Out[9]= HoldComplete[up]
```

In fact, nothing affects the arguments. (The three special symbols `Sequence`, `Evaluate`, and `Unevaluated` are all discussed in this section.)

```
In[10]:= HoldComplete[ Sequence[x, y],
                       Evaluate[0+1],
                       Unevaluated[1+1] ]

Out[10]= HoldComplete[Sequence[x, y], Evaluate[0 + 1],
  Unevaluated[1 + 1]]
```

The new attribute `HoldAllComplete` is used extensively in the *Mathematica* code for expression formatting described in Section 9.5.

■ 5.3.2 Application: Extracting Parts of Held Expressions

Let us address the problem of finding the structure of an expression inside `Hold[]` without evaluating it. We can, of course, extract the expression inside `Hold[`*expr*`]` by either `Hold[`*expr*`][[1]]` or `ReleaseHold[Hold[`*expr*`]]`. But if we use this inside `Length[]`, for example (to find the length of *expr*), then it would be evaluated, because `Length[]` does evaluate its argument. The solution is to use `MapAt[]` to wrap `Hold[]` around each element in *expr* (including the head) and then get rid of the outer `Hold[]`. The expression is now completely "frozen" and we can find its length, for example.

This expression, when given a chance to evaluate, would immediately turn into 26.

```
In[1]:= expr = Hold[ 0 1  + 2 3 +  4 5 ]

Out[1]= Hold[0 1 + 2 3 + 4 5]
```

This takes care of the head, which is not necessary here, because it is a symbol without a value.

```
In[2]:= MapAt[ Hold, expr, {1, 0} ]

Out[2]= Hold[Hold[Plus][0 1, 2 3, 4 5]]
```

`Map[]` maps `Hold` at each element of our original expression.

```
In[3]:= Map[ Hold, %, {2} ]

Out[3]= Hold[Hold[Plus][Hold[0 1], Hold[2 3], Hold[4 5]]]
```

This gets rid of the outer `Hold[]`.

```
In[4]:= frozen = %[[1]]

Out[4]= Hold[Plus][Hold[0 1], Hold[2 3], Hold[4 5]]
```

Here is its length without evaluating it.

```
In[5]:= Length[frozen]

Out[5]= 3
```

This extracts its second element without evaluating it.

```
In[6]:= frozen[[2]]

Out[6]= Hold[2 3]
```

The function `WrapHold[]` implements these steps. It is shown in Listing 5.3–2.

You would not want any of these expressions evaluated (for different reasons). This example makes a good test of `WrapHold[]` because any mistake would not go unnoticed.

```
In[7]:= WrapHold[ Exit[Quit[], 1/0, 3^10^10] ]

                                                   10
                                        1        10
Out[7]= Hold[Exit][Hold[Quit[]], Hold[-], Hold[3      ]]
                                        0
```

The built-in function `Extract[]` implements some of the functionality presented here directly. Its third argument is the head to be wrapped around the result.

```
In[8]:= Extract[expr, {1,2}, Hold]

Out[8]= Hold[2 3]
```

```
WrapHold::usage = "WrapHold[expr] wraps Hold[] around the head
    and the elements of expr without evaluating them."

Begin["`Private`"]

SetAttributes[WrapHold, HoldAll]

WrapHold[expr_] :=
    Map[ Hold, MapAt[Hold, Hold[expr], {1, 0}], {2}] [[1]]

End[];
```

Listing 5.3–2: WrapHold.m: Wrap `Hold[]` around all parts of an expression

The functions `Edit[]`, `EditIn[]` and `EditDefinition[]` that should be available in most versions of *Mathematica* (they are not built-in) make heavy use of such ideas.

5.3.3 Evaluating Arguments That Are Normally Not Evaluated

If a function has the attribute `HoldAll`, then its arguments are used inside the function body without prior evaluation, as we have just seen. Evaluation can be forced in such a case by using `Evaluate[]`.

This measures the time it takes to sum the integers from 1 to 10000. The `Sum[]` is evaluated only inside the function `Timing[]`.

```
In[9]:= Timing[ Sum[i, {i, 10000}] ]
Out[9]= {0.47 Second, 50005000}
```

In this case, the sum is evaluated before it is passed to `Timing[]`. `Timing[]` then receives the integer 50005000 as argument and it takes almost no time to evaluate it again.

```
In[10]:= Timing[ Evaluate[Sum[i, {i, 10000}]] ]
Out[10]= {0. Second, 50005000}
```

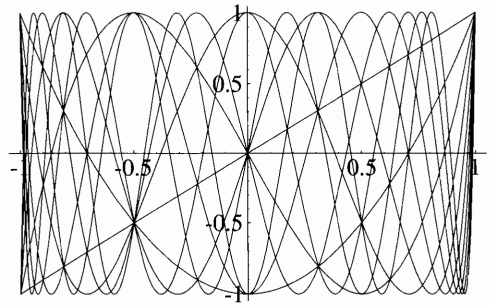
Sometimes `Evaluate[]` is necessary. The command `Plot[]`, for example, does not evaluate its first argument. It looks at the unevaluated argument to see whether it is a list, in which case it prepares to plot several functions in one picture. If it is not a list, it assumes that it is a single function. If you have a list of expressions stored as a value of some variable and you want to use it in `Plot[]` to plot all of these expressions in one picture, evaluating it (to a list of expressions) before `Plot[]` sees it is essential.

This generates a list of the first 10 Chebyshev polynomials.

```
In[1]:= Short[ Table[ChebyshevT[i, x], {i, 1, 10}], 4 ]
Out[1]//Short=
               2           3          2      4
 {x, -1 + 2 x , -3 x + 4 x , 1 - 8 x  + 8 x ,

           3       5
  5 x - 20 x  + 16 x , <<2>>, 1 + <<4>>,

            3        5        7        9
  9 x - 120 x  + 432 x  - 576 x  + 256 x ,

            2         4          6          8         10
  -1 + 50 x  - 400 x  + 1120 x  - 1280 x  + 512 x  }
```

Here is a picture of them. It would not have worked in the form `Plot[%, {x, -1, 1}]`.

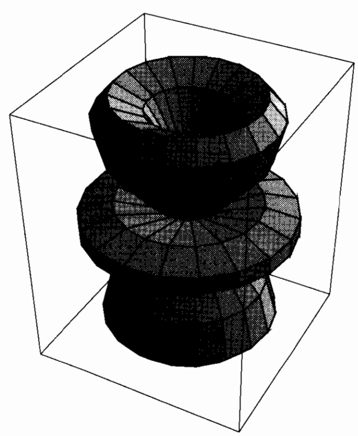
```
In[2]:= Plot[ Evaluate[%], {x, -1, 1} ];
```



Sometimes `Evaluate[]` is merely a matter of efficiency. For an example, let us look at `SphericalPlot3D[]` from the package ParametricPlot3D.m. `SphericalPlot3D[]` does not evaluate its first argument but passes it on to `ParametricPlot3D[]` which does not evaluate it either. It is finally evaluated inside the code of `ParametricPlot3D[]`. If we wanted to generate a spherical plot of the absolute value of a spherical harmonic, using the function `Abs[SphericalHarmonicY[3, 1, theta, phi]]`, then that function is evaluated over and over again. `SphericalHarmonicY[]` does evaluate to a polynomial in trigonometric functions, however, and in this form would take much less time to evaluate numerically.

Generating this plot takes only about 20% of the time that it would take without `Evaluate[]`.

```
In[1]:= SphericalPlot3D[ Evaluate[
          Abs[SphericalHarmonicY[3, 1, theta, phi]]],
         {theta, 0, Pi}, {phi, 0, 2Pi} ];
```



This is the expression into which $Y_1^3(\theta, \varphi)$ evaluates.

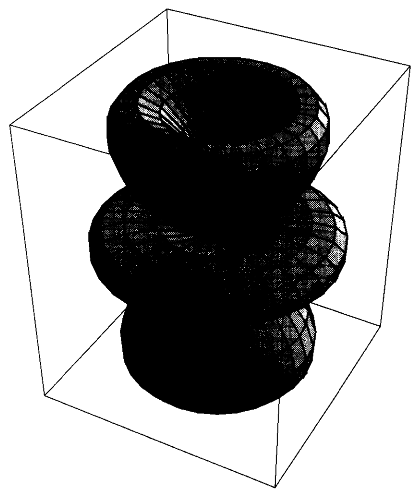
```
In[2]:= SphericalHarmonicY[3, 1, theta, phi]

            I phi       21                      2
         -(E      Sqrt[--] (-1 + 5 Cos[theta] ) Sin[theta])
                        Pi
Out[2]= ---------------------------------------------------
                                  8
```

By noting the fact that both `theta` and `phi` are real-valued and by approximating the numeric quantities, we can plot it even faster.

```
In[3]:= N[ ComplexExpand[Abs[%]] ]

                                              2 2
Out[3]= 0.32318 Sqrt[(-1. + 5. Cos[theta] ) ]

                    2
   Sqrt[Sin[theta] ]
```

We use the saved time to increase the plot resolution.

```
In[4]:= SphericalPlot3D[ Evaluate[%],
          {theta, 0, Pi}, {phi, 0, 2Pi},
          PlotPoints -> {36, 30} ];
```



■ 5.3.4 Passing Unevaluated Arguments to Ordinary Functions

The previous subsection showed how the evaluation of arguments of functions that do not evaluate arguments can be forced. Here, we look at the complementary problem: how to prevent evaluation of arguments of ordinary functions. The wrapper `Unevaluated[`*expr*`]` prevents the evaluation of *expr* when it appears as argument of a function. The unevaluated form of *expr* is passed to the function (with `Unevaluated[]` stripped).

Normally, `Length[]` evaluates its argument. In this case, however, it receives the unevaluated expression `Plus[1, 2, 3, 4]` as argument. Its length is 4.

```
In[5]:= Length[ Unevaluated[1+2+3+4] ]
Out[5]= 4
```

One use of `Unevaluated[]` is for preserving arguments of functions in unevaluated form. If a function has the attribute `HoldAll`, its arguments are not evaluated and the unevaluated forms are substituted for all occurrences of the argument in the body of the function (see Section 5.3.1). If such an argument appears inside an ordinary function (which does not keep its arguments unevaluated), it would be evaluated at that point. If this evaluation is undesirable, you can wrap the argument in `Unevaluated[]`. Here is the template for this use:

```
SetAttributes[f, HoldAll]

f[a_, ...] :=
    Module[{...},
        ⋮
        x = g[Unevaluated[a]];
        ⋮
    ]
```

Maintaining arguments in unevaluated form

In this example, the argument a is passed unevaluated to the function g.

■ 5.3.5 Sequences

When an expression of the form f[arg_1, arg_2, ...] is evaluated, the value of each of the arg_i becomes one element of the resulting expression. If, however, the value matched by a pattern of the form *name*__ or *name*___ is used in the position of one of the arg_i, then it is spliced in, occupying as many positions as there were elements in the matched expression. The object that causes its elements to be spliced in must, of course, be an expression itself, because all internal objects are expressions. The head of it is the symbol Sequence.

An example: The expression f[a, b, c, d] matches the pattern f[x_, opts___] with x becoming a as usual and opts becoming Sequence[b, c, d]. If opts is then used as an element of another expression, the sequence goes away and its elements are spliced in. So the expression g[u, opts, v] becomes g[u, b, c, d, v]. In the special case f[a] in which the matched sequence is empty, opts just becomes Sequence[]. If used as an element, this element simply goes away: g[u, opts, v] becomes g[u, v]!

An expression with head Sequence is rather elusive. You cannot even look at it with FullForm[].

Here is an empty sequence.

```
In[1]:= Sequence[]

Out[1]= Sequence[]
```

The argument of FullForm[] goes away, giving a syntax error.

```
In[2]:= FullForm[%]

FullForm::argx:
   FullForm called with 0
     arguments; 1 argument is expected.
Out[2]= FullForm[]
```

About the only way to create a sequence of certain elements is to first create a list of these elements and then replace the head List by Sequence using the function Apply[*newhead*, *expression*]. In most cases, we prefer to write it in infix form as *newhead* @@ *expression*. We used this method in the function FilterOptions[] in Section 3.2.4.

Splicing of sequences happens before any other evaluation. There is an attribute SequenceHold that prevents the splicing. The attribute HoldAllComplete, described at

the end of Section 5.3.1, also prevents sequence splicing. These attributes can be used if you need to write rules for the explicit manipulation of sequences.

Here is an attempt to define a function that takes a sequence as argument.

```
In[3]:= f[a_Sequence] := {a}
```

It does not work as expected, because the sequence is spliced in before our rule takes effect.

```
In[4]:= f[Sequence[x, y]]
Out[4]= f[x, y]
```

It is necessary to give the function the attribute `SequenceHold`.

```
In[5]:= SetAttributes[g, SequenceHold];\
        g[a_Sequence] := {a}
```

Now it works. The sequence is still spliced in, but only after the rule has taken effect.

```
In[6]:= g[Sequence[a, b]]
Out[6]= {a, b}
```

We shall use `SequenceHold` in our timing function in Section 8.1.2.

5.4 Nonlocal Flow of Control

Good programming style dictates that the program's flow of control should be structured using iteration, repetition, and recursion, rather than unstructured jumping around using `GoTo[]`. There is one situation which structured flow of control cannot handle well: exceptional exit out of deeply nested function calls or loops. Such exits happen in the event of error conditions that can be detected only at run time and typically involve a failure of an external operation (such as opening a file that does not exist), improper interactive input, or the failure of an internal algorithm (such as the impossibility of solving a certain list of equations).

5.4.1 Breaking Out of Repetitions

The command `Break[]` exits the nearest enclosing `Do`, `For`, or `While` loop. It is typically invoked inside a condition of the form `If[`*test*`, Break[]]`.

An example of the proper use of `Break[]` is the iterative version of binary search. The function `BinarySearch[`*list*`, `*elem*`]` returns the index of element *elem* in the ordered list *list*, or 0 if *elem* does not occur in *list*.

```
BinarySearch::usage = "BinarySearch[list, elem] finds the position of elem
    in the sorted list. If elem does not accur in list, 0 is returned."

BinarySearch[list_, elem_] :=
    Module[{n0 = 1, n1 = Length[list], m},
        While[n0 <= n1,
            m = Floor[(n0 + n1)/2];
            If[ list[[m]] == elem, Break[] ]; (* found *)
            If[ list[[m]] < elem, n0 = m+1, n1 = m-1 ]
        ];
        If[ n0 > n1, 0, m ]
    ]
```

BinarySearch1.m

The algorithm proceeds by narrowing the part of the list that would contain the element. The variables `n0` and `n1` mark the boundaries of the part of the list still to be searched. At each iteration, the element in the middle (with index $m = (n_0 + n_1)/2$) is tested. If it happens to be the one we are looking for, we exit the loop. Otherwise we determine which half of the list, $(n_0, m - 1)$ or $(m + 1, n_1)$, would contain the element and continue the search with adjusted boundaries. After the loop, we need to find out whether the loop was exited because of the `Break[]` or because the loop condition $n_0 \leq n_1$ was no longer true. Note that the test inside the `If` is the negation of the loop condition.

If the loop occurs near the end of the body of a definition, as it does here, there is an alternative way to exit the loop: `Return[`*val*`]` to leave the loop and return from the definition at the same time. This method makes the binary search code even simpler:

```
BinarySearch::usage = "BinarySearch[list, elem] finds the position of elem
    in the sorted list. If elem does not accur in list, 0 is returned."

BinarySearch[list_, elem_] :=
    Module[{n0 = 1, n1 = Length[list], m},
        While[n0 <= n1,
            m = Floor[(n0 + n1)/2];
            If[ list[[m]] == elem, Return[m] ]; (* found *)
            If[ list[[m]] < elem, n0 = m+1, n1 = m-1 ]
        ];
        0  (* not found *)
    ]
```

BinarySearch2.m

Another important class of problems where `Break[]` is useful is the treatment of interactive input. We want to repeatedly prompt the user for input until we receive a valid answer. The code shown in Listing 5.4–1 will insist in the user typing a number that satisfies a given predicate. (A predicate is a function that returns `True` or `False`.) Such loops have their exit somewhere in the middle, so neither `While[]` nor `Until[]` (from Section 4.2.4) is quite appropriate.

```
GetNumber[prompt_String, predicate_:(True&)] :=
    Module[{answer},
        While[ True,
            answer = Input[prompt];
            If[ NumberQ[answer] && predicate[answer], Break[] ];  (* good   *)
            If[ answer === EndOfFile, Break[] ];                  (* escape *)
            Print["Please enter a number that satisfies ", predicate]
        ];
        answer
    ]
```

Listing 5.4–1: GetNumber.m: Repeatedly prompting the user for input

It is a good idea to allow some escape mechanism from the loop that prompts for input. In this case, generating an end-of-file character will exit the loop. If the answer is rejected, a friendly message is printed, and another prompt is printed. The default value for the predicate is the pure function `True&` that returns `True` independent of its argument, and so it will accept any number.

An input that does not satisfy the predicate will issue a message and prompt again.

```
In[1]:= input = GetNumber["Enter a prime: ", PrimeQ]

Enter a prime: 9
Please enter a number that satisfies PrimeQ
Enter a prime: 11

Out[1]= 11
```

Mathematica can read any expression. You can perform arbitrary calculations in the input you type. If it does evaluate to a number, it will be accepted.

```
In[2]:= GetNumber["Enter any number: "]

Enter any number: (-1 + 5 I)^2

Out[2]= -24 - 10 I
```

■ 5.4.2 Catch and Throw

A "pure" programming language, such as PASCAL, that does not offer nonlocal flow of control makes it hard to recover from error conditions encountered deep inside nested functions or loops. One cumbersome solution is to use a Boolean variable that is set to `True` if an error is encountered; this variable is then tested in all enclosing loops.

A few programming languages, including STANDARD ML and JAVA, offer *exceptions*. Exceptions allow you to raise an error condition at any point in a program and to test for the occurrence such an error in an enclosing piece of program.

Mathematica offers this recovery mechanism with `Catch[]` and `Throw[]`. Whenever an error condition occurs, the current computation is abandoned with `Throw[`*errval*`,` *tag*`]`. At an appropriate place enclosing this computation the presence of the error can be detected with *res* = `Catch[`*computation*`,` *tag*`]`. If no error occurred, the variable *res* will contain the normal result of *computation*, but if `Throw[]` was invoked, its value will be *errval*. The tags can be arbitrary expressions; they are used to bind matching pairs of `Throw[]` and `Catch[]` together to ensure that each instance of `Catch[]` handles only those invocations of `Throw[]` that it was designed to handle.

■ 5.5 Definitions

Because *Mathematica* is rule based, definitions play an important role and exist in many variations. This section treats the finer points of definitions and advanced uses.

■ 5.5.1 Different Forms of Assignment

Mathematica has two operators for assigning values to symbols and expressions. The two forms are *lhs* = *rhs* and *lhs* := *rhs*. The first form, whose internal representation is `Set[`*lhs*`, `*rhs*`]`, evaluates the right side before the assignment takes place. In the second form, internally represented as `SetDelayed[`*lhs*`, `*rhs*`]`, the right side is assigned unevaluated. Evaluation takes place only later when the rule is used. This different behavior is achieved quite simply with the use of attributes.

`Set[]` does evaluate its second argument in the usual way. (More about the evaluation of its first argument in Section 5.5.5.)

```
In[1]:= Attributes[Set]
Out[1]= {HoldFirst, Protected, SequenceHold}
```

`SetDelayed[]` does not evaluate any of its arguments.

```
In[2]:= Attributes[SetDelayed]
Out[2]= {HoldAll, Protected, SequenceHold}
```

Subection 2.4.8 of the *Mathematica* book contains examples that explain when to use which of the two forms for assignment.

Delayed assignment is most often used when the left side contains patterns. This is the usual case with global transformation rules (see Chapter 6) and *Mathematica*'s equivalent for procedure or function definitions of traditional programming languages.

Assignments to symbols are usually not delayed. But there are two interesting uses of delayed assignment for symbols that we want to mention in the next two subsections.

■ 5.5.2 Multiple Assignments

It is possible to perform multiple assignments in one statement. Typically this looks like sym_1 = sym_2 = *expr*. This assigns the value of *expr* to both sym_1 and sym_2. Internally, it is represented as `Set[`sym_1`, Set[`sym_2`, `*expr*`]]`. The inner `Set[]` is evaluated first. Besides assigning the value of *expr* to sym_2 it also returns this value which is then assigned to sym_1.

With delayed assignments, it is quite a different case. Because the right side of a delayed assignment is not evaluated, the effect of sym_1 := sym_2 := *expr*, or in internal form `SetDelayed[`sym_1`, SetDelayed[`sym_2`, `*expr*`]]`, is to assign the statement sym_2 := *expr* to sym_1. Nothing is assigned to sym_2 at this point. When sym_1 is evaluated later on the result of this evaluation would be to assign the unevaluated *expr* to sym_2. The value of the

`SetDelayed[]` function itself is `Null`, because it cannot return anything meaningful. So the value of sym_1 is `Null` which is not printed.

The value of the outer assignment is `Null`, which is not printed.

```
In[1]:= a := b := e
```

We have just defined this value for `a`.

```
In[2]:= ?a
Global`a

a := b := e
```

Using `a` performs the assignment to `b` and returns `Null`.

```
In[3]:= a
```

Now, `b`, too, has a (delayed) value.

```
In[4]:= b
Out[4]= e
```

The mixed form sym_1 := sym_2 = *expr* is of some value when used for rules with patterns on the left side. See the following Section 5.5.3 and the section on dynamic programming (2.4.9) of the *Mathematica* book.

■ 5.5.3 Application: A File of Commands

Another use of delayed assignment for symbols is for assigning complicated commands to a number of symbols in a file. You can then read in that file and execute any of these commands by simply typing the corresponding symbol. You can combine this with multiple assignment, so that once a computation has been performed, its result is stored in a variable and not recomputed thereafter.

sym := *cmd*	execute *cmd* every time the symbol *sym* is used
sym := *sym* = *cmd*	execute *cmd* the first time *sym* is used; thereafter, use the stored result

Delayed assignments to symbols

The file BookPictures.m uses this idea. It has definitions for all chapter-opener pictures in this book. Its outline is as follows (the file is shown completely in Listing 10.3–1):

```
Needs["Graphics`ComplexMap`"]
Needs["Graphics`ParametricPlot3D`"]
:

chapter1 := chapter1 =
PolarMap[ (2# - I)/(# - 1 + 0.1I)&, {0.001, 5.001, 0.25}, {0, 2Pi, Pi/15},
        Framed -> True, PlotPoints -> 40 ]
```

```
chapter2 := chapter2 =
ParametricPlot3D[{r*Cos[phi] - (r^2*Cos[2*phi])/2,
                 -(r*Sin[phi]) - (r^2*Sin[2*phi])/2,
                 (4*r^(3/2)*Cos[(3*phi)/2])/3},
                {r, 0.0001, 1}, {phi, 0, 4Pi}, PlotPoints -> {8, 60}]
⋮
```

Part of BookPictures.m

First, all necessary auxiliary definitions are made and then a single assignment is defined for each picture. An immediate assignment would compute all the graphics when you read in the file, which would take hours to do. Set up like this, each picture is computed only when you want to see it, but once a picture has been computed it is assigned to the corresponding variable; therefore, any future references to it will not recompute the picture.

In *Mathematica* versions with the *notebooks* interface, there is a different way to achieve this effect. Simply put the commands into individual input cells that are not marked as initialization cells. To evaluate one of them, select it and then choose the Kernel ▷ Evaluation ▷ Evaluate Cells menu item (or hit ENTER). The auxiliary definitions at the beginning should be put into initialization cells. The notebook BookPictures.nb has been created from BookPictures.m in this way.

■ 5.5.4 Local Definitions

Many programming languages allow the declaration of local functions and procedures. *Mathematica* is no exception; a local function declaration looks like this:

```
f[ ... ] :=
    Module[{g, ...},
        g[...] := ...;
        ⋮
    ]
```

Local function declarations

The local definition of functions within functions does not cause any performance penalty in compiled languages, such as PASCAL. The situation is different in *Mathematica*. The definition of the local function `g` inside `f` in the example above is performed *every time* `f` is called. For this reason, auxiliary functions should usually be declared at the package level, in the implementation part.

Some languages, most notably **C** and FORTRAN, do not allow local functions. In **C**, functions can be declared `static`. A static function is an auxiliary function, invisible outside its compilation unit. The direct equivalents of static functions are functions defined in the implementation part of a package, but not exported in the definition part, such as `g` in this example:

```
BeginPackage[...]
f::usage = "..."
Begin["`Private`"]
f[...] := ...; (* exported function *)
g[...] := ...; (* auxiliary, static function *)
End[]
EndPackage[]
```

Auxiliary functions in *Mathematica*

There are two cases, however, where local definitions within functions should be used: for local functions with many parameters and for dynamic programming.

■ 5.5.4.1 Sharing Variables with Local Functions

Consider this example:

```
f[ x_, y_, ... ] :=
    Module[{g, c, z},
        c = ...;
        g[z_] := c x z;
        res = g[y];
        ⋮
    ]
```

Sharing variables with a local function

The body of the local function `g` depends not only on its parameter `z`, but also on the parameter `x` of the outer function `f` and on one of its local variables, `c`. If local definitions were not available, extra parameters would be needed for these additional dependencies:

```
g[z_, c_, x_] := c x z
f[ x_, y_, ... ] :=
    Module[{c},
        c = ...;
        res = g[y, c, x];
        ⋮
    ]
```

Parameters instead of local variables

The introduction of many additional parameters can lead to performance penalties. In fact, if `g` is called many times inside `f`, we could gain some more performance by inserting the *value* of `c` into the body of `g` like this:

```
f[ x_, y_, ... ] :=
    Module[{g, c, z},
        c = ...;
        With[{c = c},
            g[z_] := c x z
        ];
        res = g[y];
        ⋮
    ]
```

Inserting values of local variables into function bodies

This use of `With[]` is explained in Section 5.6.1.

The value of the pattern variable `x` is already inserted into the body of `g`. The construct `f[x_, ...] := With[{x = x}, g[z_] := ...]` would not work.

■ 5.5.4.2 Example: Faster Swinnerton-Dyer Polynomials

We saw in Section 4.7.5 that the computation of the Swinnerton-Dyer polynomials $s_n(x)$ according to Equation 4.7–2 leads to huge intermediate expressions. The same polynomials can alternatively be computed recursively:

$$\begin{aligned} s_0(x) &= x \\ s_n(x) &= s_{n-1}\left(x+\sqrt{p_n}\right) s_{n-1}\left(x-\sqrt{p_n}\right) . \end{aligned} \qquad (5.5\text{–}1)$$

If the intermediate expressions are expanded at each stage, the buildup of huge intermediate expressions is avoided. Because s_{n-1} is used twice, we should compute it once and define it as a local function. Listing 5.5–1 shows the fast version of our code to compute Swinnerton-Dyer polynomials. Note that the local function `sd` is defined using immediate assignment (`=`). Had we used delayed assignment (`:=`) it would, in fact, be computed twice! Remembering the results of recursive subcalculations is often termed *dynamic programming*.

The fifth Swinnerton-Dyer polynomial could not be computed with the slow code in SwinnertonDyer1.m (Listing 4.7–1); the recursive version, however, takes only about a second.

```
In[1]:= SwinnertonDyerP[5, x]

                                                2
Out[1]= 2000989041197056 - 44660812492570624 x  +

                        4                         6
   183876928237731840 x  - 255690851718529024 x  +

                        8                        10
   172580952324702208 x  - 65892492886671360 x   +

                       12                       14
   15459151516270592 x   - 2349014746136576 x   +

                     16                    18
   239210760462336 x   - 16665641517056 x   +

                 20                 22                24
   801918722048 x   - 26625650688 x   + 602397952 x   -

             26         28        30    32
   9028096 x   + 84864 x   - 448 x   + x
```

```
BeginPackage["ProgrammingInMathematica`SwinnertonDyer`"]

SwinnertonDyerP::usage = "SwinnertonDyerP[n, var] gives the minimal polynomial
    of the sum of the square roots of the first n primes."

Begin["`Private`"]

SwinnertonDyerP[ 0, x_ ] := x

SwinnertonDyerP[ n_Integer?Positive, x_ ] :=
    Module[{sd, srp = Sqrt[Prime[n]]},
        sd[y_] = SwinnertonDyerP[n-1, y];
        Expand[ sd[x + srp] sd[x - srp] ]
    ]

End[ ]

EndPackage[]
```

Listing 5.5–1: SwinnertonDyer.m: Fast computation of Swinnerton-Dyer polynomials

■ 5.5.5 Evaluation of the Left Side of an Assignment

In a rule of the form *lhs* `->` *rhs* or *lhs* `:>` *rhs*, the left side *lhs* is evaluated in the normal way, like any other expression. In a definition of the form *lhs* `=` *rhs* or *lhs* `:=` *rhs* things are different. In some applications, the exact way of evaluation of the left side is important. See also Subsection B.4.2 of the reference guide. In a definition like f`[`e_1`,` e_2`, ...,` e_n`] :=` *rhs*, the left side f`[`e_1`,` e_2`, ...,` e_n`]` is evaluated as follows. The arguments e_1, e_2, ..., e_n of the top-level function f are evaluated in the normal way. The top-level function itself is *not* evaluated, that is, no built-in code for f and no user-defined rules are applied. The definition is then made for this partially evaluated expression.

To look at the evaluated left side of the rule, we look at the internal form of the rules attached to f with `FullForm[DownValues[`*f*`] /. (l_ :> _) :> l]`. (The substitution suppresses the right side of the rules in the list of down values.) Here is an example.

We define a rule.

```
In[1]:= f[ x_ - y_ ] := g[x, y]
```

The subtraction in the left side has been replaced by addition and multiplication by -1.

```
In[2]:= FullForm[DownValues[f] /. (l_ :> _) :> l]
Out[2]//FullForm=
 List[HoldPattern[f[Plus[Pattern[x, Blank[]],
     Times[-1, Pattern[y, Blank[]]]]]]]
```

If the evaluation of an argument of the left side is undesirable, it can be suppressed by enclosing that argument in `HoldPattern[]`. A definition used in Section 7.2.3 has the basic form

```
N[Sum[args__], prec_] := NSum[args, AccuracyGoal -> prec].
```

In this form it would not work. The first argument in the left side is `Sum[args__]`. When this is evaluated it turns into `args__` and the definition would be made for `N[args__, prec_]` (which does not work). Functions do not do anything special if their arguments happen to be patterns. `Sum[args__]` is therefore treated like `Sum[`*e*`]` with *one* argument which evaluates to *e*. Therefore we have to use

```
N[HoldPattern[Sum[args__]], prec_] := NSum[args, AccuracyGoal -> prec]
```

which does not try to evaluate its first argument. The tag `HoldPattern[]` does not interfere with pattern matching.

A sum with no iterators evaluates to the summand.

```
In[3]:= Sum[ term ]

Out[3]= term
```

`Sum[args__]` has been turned into `args__` and no rule can be defined for `Sum`.

```
In[4]:= N[Sum[args__], prec_] :=
          NSum[args, AccuracyGoal -> prec]

N::nosym: N[args__, prec_]
     does not contain a symbol to which to attach a rule.

Out[4]= $Failed
```

Now it works.

```
In[5]:= N[HoldPattern[Sum[args__]], prec_] :=
          NSum[args, AccuracyGoal -> prec]
```

`HoldPattern[]` can also be used on the left side of a rule which is otherwise evaluated fully. To prevent any evaluation of the left side of a rule you can use

expr `/. HoldPattern[`*lhs*`] ->` *rhs*.

We used this form in Section 5.2.4, for example.

Note that *Mathematica* uses `HoldPattern[]` by itself on lists of downvalues. This is done to protect them from evaluation after they have been defined.

```
In[6]:= DownValues[f]

Out[6]= {HoldPattern[f[(x_) - (y_)]] :> g[x, y]}
```

5.6 Advanced Topic: Scopes of Names

The concepts of *scope* and *local variables* were introduced in Section 4.1. In a compiled language, scopes of symbols are usually figured out at compile time. Symbols are not created at run time. In *Mathematica* any new name a user types in defines a new symbol. Rules for scoping are much more complicated in these cases. Another difference is the fact that in *Mathematica* symbols can stand for themselves. In this way, symbols (values of parameters, for example) could be brought into the scope of a declaration that declares the same symbol as a local variable.

A major step from version 1.2 to version 2.0 was the introduction of *lexical scoping*. In a functional language where you can write deeply nested expressions, scoping of symbols is important to isolate a program from the effects of coincidences of local variables and arguments of functions.

There are three areas where scoping is used: scopes of symbols, scopes of values, and scoping for pattern variables.

5.6.1 Scopes of Symbols

Section 2.6 of the *Mathematica* book discusses these issues in some detail. Let us give some more examples here. There are three constructs that introduce local symbols.

`Module[{`x_1`, ..., `x_n`}, `*body*`]`	local variables
`With[{`x_1`=`v_1`, ..., `x_n`=`v_n`}, `*body*`]`	local constants
`Function[{`x_1`, ..., `x_n`}, `*body*`]`	local parameters

Declaring local symbols

In each case, the scope of the declared symbols $x_1, \ldots, x_n$ is *body* (which can be any expression, usually a sequence of statements). Whenever one of these symbols would be brought into the scope of such a declaration, appropriate steps are taken to make sure that the symbol declared as local is indeed treated as distinct from the symbol introduced. One way of crossing the boundaries of a scope is through evaluation and function application.

We assign the symbol `x` to `y`. Evaluating `y` inside a scope where `x` is declared local would bring this outside `x` into the scope.

```
In[1]:= y = x
Out[1]= x
```

The `x` occurring literally in the body of this `Module[]` is the one that is treated as local. The `x` that is the value of `y` is treated as completely distinct. It is not affected by the assignment.

```
In[2]:= Module[ {x}, x = 5; x + y ]
Out[2]= 5 + x
```

Here, too, only the x literally present inside the body of With[] is replaced by 5.

```
In[3]:= With[ {x = 5}, x + y ]
Out[3]= 5 + x
```

Applying this pure function to the argument 5 is done in the same way.

```
In[4]:= Function[x, x + y][5]
Out[4]= 5 + x
```

Set up in this way, the name given to a local symbol is completely irrelevant. The meaning of the program does not depend on any particular choice.

Another way of potentially crossing scope boundaries is through nesting of these scoping constructs.

The outer With[] replaces every occurrence of y within its body by x. The x declared in the inner Module[] is treated as different.

```
In[5]:= With[ {y = x}, Module[{x}, x = 5; x + y] ]
Out[5]= 5 + x
```

If you use a different symbol in the Module[] it should be clear that the outer oix is not affected. Because the name of a local symbol does not matter, you should get the same behavior as in the preceding example.

```
In[6]:= With[ {y = x}, Module[{xx}, xx = 5; xx + y] ]
Out[6]= 5 + x
```

You could always get the desired behavior by choosing *new* symbols in all local declarations. When you use these scoping constructs, however, you do not have to worry about this. If you write a program for others to use, you never know what names users will choose for their variables. With these scoping constructs their choices do not matter. As a matter of fact, the implementation of Module[] does create a new symbol every time it is called. The names of these symbols are derived from the name used in the declaration. See the *Mathematica* book for an explanation of how this works.

This semantics of scoping constructs is termed *static binding* or *lexical scoping*. With static binding it is possible to tell from the program text alone which occurrences of a symbol belong to which declaration.

The concepts of scoping may be quite subtle, as this program fragment shows:

```
x = 2;
With[{x = x},
    f[y_] := 2 x y
]
```

To understand this program, first note that the initializer *val* in With[{*var* = *val*}, *body*] is *not* in the scope of *var*. Therefore, the *local* variable x is initialized by the value of the *global* variable x, that is, by 2. This value is then inserted into the body of the definition of f. There are two reasons to do this, instead of defining f in the usual way, not enclosed in With[]:

- For efficiency reasons. Accessing the value 2 directly, rather than first determining the value of x, is (slightly) faster.

- To guard against any future modification of `x`. We want the current value of `x` to be "frozen" inside the body of `f`.

The following computations show the difference:

We set the global variable `x` to 2.	`In[7]:= x = 2;`
Here is an ordinary definition for `f1`.	`In[8]:= f1[y_] := 2 x y`
This definition of `f2` freezes the current value of `x` in the body of `f2`.	`In[9]:= With[{x = x}, f2[y_] := 2 x y]`
To show the differences, we reset `x`.	`In[10]:= x = 99;`
The two results differ. The first one uses the *current* value of `x`, the second one uses the value `x` had when `f2` was *defined*.	`In[11]:= {f1[10], f2[10]}` `Out[11]= {1980, 40}`

In the function `f1`, the variable `x` is dynamically scoped; in `f2` it is lexically (statically) scoped.

■ 5.6.2 Scopes of Values

Besides introducing local symbols, distinct from any other symbols, we can also localize just the values of symbols.

`Block[{`x_1`, ..., `x_n`}, `*body*`]`	local values
`Do[`*body*`, {`*i*`, ...}]`	local iterator variable
`Table[`*expr*`, {`*i*`, ...}]`	local iterator variable in a table
`Sum[`*expr*`, {`*i*`, ...}]`	local iterator variable in a sum
`Product[`*expr*`, {`*i*`, ...}]`	local iterator variable in a product

Declaring local values

The scoping construct `Block[]` introduces local variables in a similar way as `Module[]` does, but it localizes only the *values* of these variables, not the symbols themselves. This is sufficient to do local computations without affecting any global values that a local variable might have. It does not protect against conflicts of names, though.

We assign a global value to `z`.	`In[1]:= z = 17` `Out[1]= 17`
Inside the block `z` behaves like a variable without an initial value.	`In[2]:= Block[ {z}, z = 5; z ]` `Out[2]= 5`

The global value of z has not been disturbed.

```
In[3]:= z
Out[3]= 17
```

Now we try out the example given earlier for `Module[]`. We assign the symbol `x` to `y`.

```
In[4]:= y = x
Out[4]= x
```

This time the `x` that is the value of `y` is captured by the scoping construct. It is not treated as distinct from the `x` declared in `Block[]`.

```
In[5]:= Block[ {x}, x = 5; x + y ]
Out[5]= 10
```

Again, the global value of `x` (none in this case) is not affected.

```
In[6]:= x
Out[6]= x
```

It is usually better to use `Module[]` than `Block[]` unless you need the particular behavior of `Block[]`. One application is to temporarily change the value of a *system variable*, without affecting its global setting. We use this in Section 8.1.2 (with `$Pre`), in Section 10.2.1 (with `$DisplayFunction`), and in Section 7.2.4 (with `$MaxPrecision`). In such a case you cannot use `Module[]` because this would create a new symbol distinct from the variable that *Mathematica* knows about. Another application is to temporarily disable the infinite recursion test built into the evaluator. You can write a procedure like this:

```
proc[ x_, y_, ... ] :=
    Block[{$RecursionLimit = Infinity},
    Module[{a, b, ...},
        ⋮
        (* some very recursive computation *)
        ⋮
    ]]
```

Temporarily setting the recursion limit to ∞

Within the body of this procedure the value of the *system variable* `$RecursionLimit` is set to Infinity. This is the value that the evaluator uses.

The mechanism used by `Block[]` to localize the value of a variable is also used by iterators to localize the value of iterator variables.

We give a global value to the variable `i`.

```
In[1]:= i = 17
Out[1]= 17
```

The global value of `i` does not disturb the iterator.

```
In[2]:= Product[ x - i, {i, 1, 5} ]
Out[2]= (-5 + x) (-4 + x) (-3 + x) (-2 + x) (-1 + x)
```

The global value of `i` is not disturbed either.

```
In[3]:= i
Out[3]= 17
```

For this case, it is preferable to use an implicit `Block[]` rather than `Module[]` because we frequently bring expressions into the scope of an iterator and do not want the iterator variable treated as distinct.

Here we generate an expression that shall serve as the body in an iterator.

```
In[4]:= expr = k^2 + k
                 2
Out[4]= k + k
```

We want the `k` in `expr` to refer to the iterator variable to compute $\sum_{k=1}^{10} k^2 + k$.

```
In[5]:= Sum[ expr, {k, 1, 10} ]
Out[5]= 440
```

If *Mathematica* used an implicit `Module[]` we would get this behavior, which is rarely desired. (If it is desired, you can see here how to achieve it.)

```
In[6]:= Module[ {k}, Sum[ expr, {k, 1, 10} ] ]
                        2
Out[6]= 10 k + 10 k
```

■ 5.6.3 Scopes of Pattern Variables

Pattern variables also introduce scopes. The effect of this treatment of pattern variables is usually noticeable only if definitions and other scoping constructs (such as `Module[]`) are nested.

f[*x*_] := *body*, *f*[*x*_] = *body*	*x* is local in *body*
f[*x*_] :> *rhs*, *f*[*x*_] -> *rhs*	*x* is local in *rhs*

Scopes of pattern variables

A rule of the form `f[x_] :> x^2` introduces a local symbol—the pattern variable `x`. See Subsection 2.6.4 of the *Mathematica* book for a more detailed discussion. Here we present an example similar to the ones in Section 5.6.1. We bring symbols into the scope of a rule through nesting of scoping constructs.

The pattern variable `x` is renamed to `x$` to avoid the conflict with the value of `y`, which is also the symbol `x`.

```
In[1]:= With[ {y = x}, f[x_] :> x + y ]
Out[1]= f[x$_] :> x$ + x
```

This renaming of pattern variables does not have any effect because the names do not matter at all, as we have seen several times in the preceding subsections. Renaming of local symbols also happens in pure functions, as the following example shows.

This defines a function `Curry2` that takes a function `f` of two arguments as argument and returns a function that does the same as `f`, but takes one argument at a time.

```
In[2]:= Curry2[f_] := Function[x, Function[y, f[x, y]]]
```

The formal parameters of the two embedded pure functions have been renamed. This has been done as a precautionary measure. No conflict of names has happened so far.

```
In[3]:= Cplus = Curry2[Plus]
Out[3]= Function[x$, Function[y$, x$ + y$]]
```

This is a function that adds 5 to its argument.

```
In[4]:= plus5 = Cplus[5]
Out[4]= Function[y$, 5 + y$]
```

This was a rather complicated way of writing 5 + 7.

```
In[5]:= plus5[7]
Out[5]= 12
```

Here we see that renaming is necessary. We wanted a function that adds y to its argument, not one that doubles it (had y$ been equal to y).

```
In[6]:= plusy = Cplus[y]
Out[6]= Function[y$, y + y$]
```

There is one important exception to lexical scoping: pattern variables are not scoping during the evaluation of a rule or definition.

Eventually, we want to define a function with this expression as its body.

```
In[1]:= Expand[(1+x)^2]
                   2
Out[1]= 1 + 2 x + x
```

We get the desired result.

```
In[2]:= f[x_] = %
                   2
Out[2]= 1 + 2 x + x
```

Everything is as expected.

```
In[3]:= ?f
Global`f
f[x_] = 1 + 2*x + x^2
```

If pattern variables were scoping during definitions, this example would not have worked. In line `In[2]` we brought the expression `x^2 + 2 x + 1` into the scope of the pattern variable `x`. Following lexical scoping, the pattern variable should have been renamed. The function we would have defined in this case is

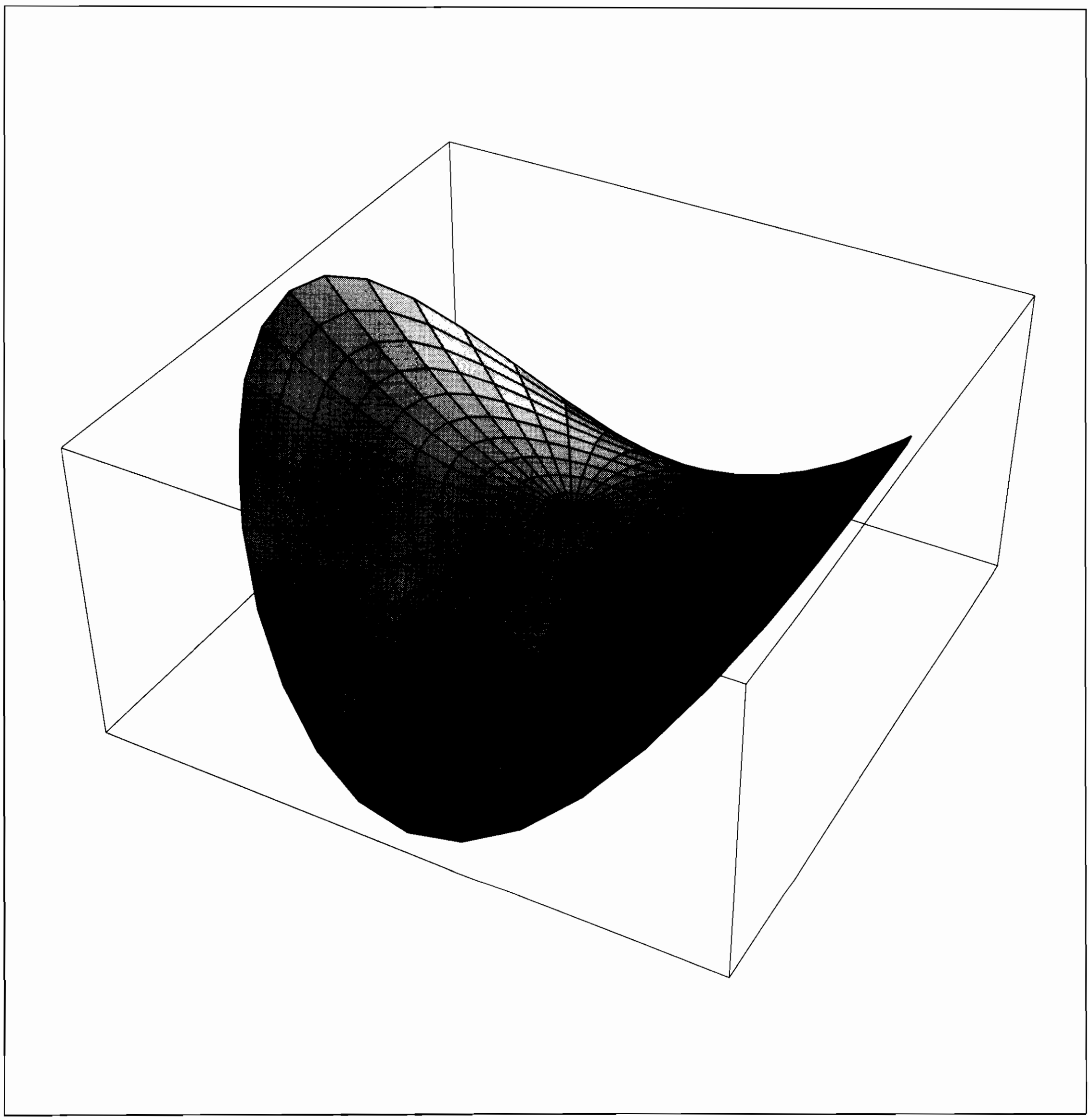
```
f[x$_] := x^2 + 2x + 1,
```

which is quite different. Because the designers of *Mathematica* anticipated this interactive way of defining functions, they chose to implement this exception.

Beware of `g[x_] := %`. It is quite different from `h[x_] = %`. Whenever you use `g`, the value of the *now* previous output is used, rather than the output at the time `g` was defined.

Chapter 6

Transformation Rules



So far we have used definitions mostly in the way that other programming languages use procedures. Now we want to use rules to define simplifications and transformations of expressions. The ability to do this easily is probably the most important aspect of *Mathematica*.

Section 1 introduces the notions of *simplification* and *normal form*. Once we have identified a normal form for a class of expressions, we can give rules to transform other expressions into this normal form. As an example, we derive rules for normal forms of odd and even functions and look at symmetries of tensors.

Section 2 looks in more detail at how to write functions that apply a set of rules to an expression. The example we examine comes from trigonometry, which is especially rich in formulae.

In the next section, we look at how to define rules that will be automatically applied to all expressions.

Knowledge of *Mathematica*'s flexible pattern matching language is important for writing efficient rule sets. In Section 4 we look at some advanced aspects of pattern matching for simplification rules and mathematical transformations.

Section 5 is about languages and grammars. In *Mathematica*, it is easy to define predicates that recognize certain classes of expressions. We explain the theory behind this and give an example.

About the illustration overleaf:

The saddle surface with a polar parameterization. In Cartesian coordinates the equation of the surface is $z = x^2 - y^2$. In cylindrical coordinates this becomes $z = r^2(\cos^2\varphi - \sin^2\varphi)$ which simplifies to $z = r^2\cos(2\varphi)$.

```
CylindricalPlot3D[r^2 Cos[2 phi], {r, 0, 1/2, 1/20}, {phi, 0, 2Pi, 2Pi/36}]
```

■ 6.1 Simplification Rules and Normal Forms

Simplification and *normal form* are two key concepts in term rewriting and rule-based programming. This section introduces these concepts and discusses a tutorial application example: odd and even functions.

■ 6.1.1 The Normal Form of Expressions

An important concept in simplifying expressions is that of a *normal form* of an expression. The number 0, for example, could be written in many different ways, as 0, $1-1$, i^2+1, or $\cos(\pi/2)$. All are equivalent and we have a clear idea about which is the simplest. Further, we know how to transform all others into the simplest form.

If we can define a normal form for a class of expressions, then two different but equivalent expressions can both be simplified to this normal form and it is therefore easy to decide whether two expressions are equivalent.

A normal form need not be the intuitively simplest form. For polynomials, the fully expanded form is a normal form. We know how to expand polynomials and if two polynomials are equal, then their expanded form is the same. Yet for many polynomials, we would consider a factored form to be simpler.

To decide whether these two polynomials are the same we expand them and compare the results.

```
In[1]:= { x^2-1, (x-1)(x+1) }

                 2
Out[1]= {-1 + x , (-1 + x) (1 + x)}
```

They are indeed equal.

```
In[2]:= Expand[ % ]

                 2         2
Out[2]= {-1 + x , -1 + x }
```

The normal—expanded—form of $(x+1)^5$ is more "complicated" than the factored form.

```
In[3]:= Expand[ (x + 1)^5 ]

                         2       3      4    5
Out[3]= 1 + 5 x + 10 x  + 10 x  + 5 x  + x
```

Here the expanded form is simpler.

```
In[4]:= Expand[ (-1 + x)(1 + x)(1 + x^2)(1 + x^4) ]

                8
Out[4]= -1 + x
```

If there are several ways of writing an expression, we should try to find one that can serve as the normal form. A necessary condition is that we have rules for reducing all other forms to this form and that the rules applied to the normal form itself do not change it. Otherwise, we would go on and on applying rules without an end.

This can be shown with *odd* and *even* functions. A function $f(x)$ is odd if $f(-x) = -f(x)$, and it is even if $f(-x) = f(x)$. Examples are the trigonometric functions. They are all either even or odd.

For an odd function f, there are two ways to write $f(x)$, either as $f(x)$ or as $-f(-x)$. We can therefore decide that the normal form for such functions should have nonnegative arguments. Here are the rules that perform these simplifications.

```
OddEvenRules = {
    (f_Symbol?OddQ)[x_?Negative]  :> -f[-x],
    (f_Symbol?EvenQ)[x_?Negative] :>  f[-x]
}
```

Simplification of odd and even functions of negative arguments

We use the standard predicates `OddQ` and `EvenQ`, defined for integers, to test functions for being odd or even. A function opf is declared odd by an assignment of the form f`/: OddQ[`f`] = True`.

Definitions for properties of symbols, such as our `OddQ[`*sym*`]` should be attached to the symbol, not the property; that is, a tag should be used: *sym*`/: OddQ[`*sym*`] = ...`.

With these rules we can perform a few trivial simplifications.

We declare `e` to be even and `o` to be odd.

```
In[2]:= e/: EvenQ[e] = True; \
        o/: OddQ[o] = True;
```

Applying the rules puts this expression into standard form. The usual arithmetic rules then perform further simplifications. In this way we easily perform the simplification $o(x) + o(-x) = 0$.

```
In[3]:= o[-2] + o[2] + e[-1] + e[1] /. OddEvenRules
Out[3]= 2 e[1]
```

No simplification is performed here because the argument `-a` does not satisfy the predicate `Negative[]`. More rules are necessary for dealing with symbolic arguments.

```
In[4]:= e[-a] /. OddEvenRules
Out[4]= e[-a]
```

For symbolic arguments we cannot tell whether they are negative or not. We therefore decide that the normal form is one where there is no *explicit* minus sign in the argument.

This is our first try at a rule for symbolic expressions. It works here.

```
In[5]:= e[-a] /. e[-x_] :> e[x]
Out[5]= e[a]
```

But it is not general enough.

```
In[6]:= e[-2a] /. e[-x_] :> e[x]
Out[6]= e[-2 a]
```

The full form of the expression `e[-2a]` and the pattern in the rule show why the pattern did not match.

```
In[7]:= FullForm[ {e[-2a], e[-x_]} ]
Out[7]//FullForm=
 List[e[Times[-2, a]], e[Times[-1, Pattern[x, Blank[]]]]]
```

This pattern is more general. It allows for any negative number in the product.

```
In[8]:= e[-2a] /. e[n_?Negative x_] :> e[-n x]
Out[8]= e[2 a]
```

By making `x_` optional, this rule can even deal with the old case of a purely numerical argument.

```
In[9]:= e[-2] /. e[n_?Negative x_.] :> e[-n x]
Out[9]= e[2]
```

The improved rules that deal with numerical and symbolic arguments are shown below.

```
OddEvenRules = {
    (f_Symbol?OddQ)[n_?Negative x_.]  :> -f[-n x],
    (f_Symbol?EvenQ)[n_?Negative x_.] :>  f[-n x]
}
```

Simplifications for negative and symbolic arguments

■ 6.1.2 Ordering of Expressions

If you type in the expression b+a, then *Mathematica* turns it into a+b. This form can hardly be considered simpler, but the built-in ordering function nevertheless provides a normal form for sums and products. This concept is quite powerful indeed. By sorting all the terms of a long sum into standard order, it is quite easy to combine terms that are the same and also perform all the numerical additions that are possible, because numbers are sorted first.

Continuing our example with odd and even functions, we can now define a normal form for such functions of arguments that are sums: The normal form shall not have a minus sign in the *first* term of the sum.

This rule works in this simple case.

```
In[1]:= o[-a + b] /. o[n_?Negative x_ + y_] :> -o[-n x - y]
Out[1]= -o[a - b]
```

But it also transforms this expression which according to our definition is already in normal form (because the first term has no minus sign).

```
In[2]:= o[a - b] /. o[n_?Negative x_ + y_] :> -o[-n x - y]
Out[2]= -o[-a + b]
```

To avoid this, we use the ordering predicate as a side condition for the rule.

```
In[3]:= o[a - b] /.
          o[n_?Negative x_ + y_] /; OrderedQ[{x, y}] :>
          -o[-n x - y]
Out[3]= o[a - b]
```

It still simplifies this expression as it should, because the first term is -a.

```
In[4]:= o[b - a] /.
          o[n_?Negative x_ + y_] /; OrderedQ[{x, y}] :>
          -o[-n x - y]
Out[4]= -o[a - b]
```

Again we can make x optional to allow for a single negative number at the beginning of the sum.

```
In[5]:= o[a - 1] /.
          o[n_?Negative x_. + y_] /; OrderedQ[{x, y}] :>
          -o[-n x - y]
Out[5]= -o[1 - a]
```

These rules do not work correctly with sums of more than two arguments. In such a case y_ in the pattern n_?Negative x_. + y_ matches the *sum* of all remaining terms. Many simple terms of the form n_?Negative x_. will be ordered before a sum of terms and our rule would match in more that one way.

The single term -b is ordered before the sum a + c and the rule matches, even though the term is already in normal form according to our definition.

```
In[6]:= o[a - b + c] /.
            o[n_?Negative x_. + y_] /; OrderedQ[{x, y}] :>
            -o[-n x - y]
Out[6]= -o[-a + b - c]
```

The solution is a bit subtle. We need to match the sequence of remaining terms with the pattern y__ and ask for the *list* of the individual terms in the sum to be ordered.

Now the rule no longer matches because the list {-b, a, c} is not ordered.

```
In[7]:= o[a - b + c] /.
            o[n_?Negative x_. + y__] /; OrderedQ[{x, y}] :>
            -o[-n x - Plus[y]]
Out[7]= o[a - b + c]
```

It still matches this case as it should.

```
In[8]:= o[-a + b - c] /.
            o[n_?Negative x_. + y__] /; OrderedQ[{x, y}] :>
            -o[-n x - Plus[y]]
Out[8]= -o[a - b + c]
```

Listing 6.1–1 shows the final version of the rules in OddEvenRules.m.

```
OddEvenRules = {
    (f_Symbol?OddQ)[n_?Negative x_.] :> -f[-n x],
    (f_Symbol?OddQ)[n_?Negative x_. + y__] /; OrderedQ[{n x, y}] :>
        -f[-n x - Plus[y]],
    (f_Symbol?EvenQ)[n_?Negative x_.] :> f[-n x],
    (f_Symbol?EvenQ)[n_?Negative x_. + y__] /; OrderedQ[{n x, y}] :>
        f[-n x - Plus[y]]
};
```

Listing 6.1–1: OddEvenRules.m: Normal forms for arguments of odd and even functions

■ 6.1.3 Symmetries

Symmetries of functions and tensors or matrices are another source of expressions that can be written in many equivalent forms. Transformation into normal form is again the easiest way to simplify complicated expressions involving such terms.

A function of two arguments f is *symmetric* if $f(x, y) = f(y, x)$. The same definition applies also to matrices: a matrix (m_{ij}) is symmetric if $m_{ij} = m_{ji}$. The built-in ordering predicate OrderedQ of expressions can be used to decide which one of the two forms $f(x, y)$ or $f(y, x)$ to transform into the other. A rule that puts symmetric objects into normal form, therefore, looks like this:

f[x_, y_] /; !OrderedQ[{x, y}] :> *f*[y, x].

Here we define the rule that simplifies the symmetric function f.

```
In[9]:= symmetric =
            f[x_, y_] /; !OrderedQ[{x, y}] :> f[y, x];
```

The term f[b, a] is turned into f[a, b]. Standard evaluation then combines the resulting terms.

```
In[10]:= 2f[a, b] - f[b, a] //. symmetric
Out[10]= f[a, b]
```

A function g is *antisymmetric* if $g(x, y) = -g(y, x)$. The corresponding rule is

g[x_, y_] /; !OrderedQ[{x, y}] :> -*g*[y, x].

A consequence of antisymmetry is that $g(x, x) = 0$. This simplification is not entailed by the rule above. You need the additional rule *g*[x_, x_] :> 0.

Here we define a rules to simplify the antisymmetric function g.

```
In[11]:= antisymmetric = {
           g[x_, y_] /; !OrderedQ[{x, y}] :> -g[y, x],
           g[x_, x_] :> 0 };
```

The term g[b, a] is turned into -g[a, b] and g[a, a] is simplified to 0. Standard evaluation then combines the resulting terms.

```
In[12]:= 2g[a, b] - g[b, a] + g[a, a] //. antisymmetric
Out[12]= 3 g[a, b]
```

Symmetries and antisymmetries are most often expressed by global definitions, not by rules; see Section 6.3.2.

■ 6.2 Application: Trigonometric Simplifications

A good source of examples for simplification rules is *trigonometry*. There are many identities between expressions involving the trigonometric functions `Sin[]`, `Cos[]`, and `Tan[]`. Note that the simplifications discussed in Section 6.1 are performed automatically for trigonometric functions.

All these expressions have a minus sign in the first position. The cosine is even; the other functions are odd.

```
In[1]:= {Sin[-1], Cos[-x], Tan[b - a]}
Out[1]= {-Sin[1], Cos[x], -Tan[a - b]}
```

Trigonometric functions satisfy a number of further identities that we shall look at next.

■ 6.2.1 Expansion of Products and Powers

In this section, we want to use identities, such as

$$\sin x \sin y = \frac{\cos(x-y)}{2} - \frac{\cos(x+y)}{2}, \qquad (6.2\text{–}1)$$

to linearize products and powers of trigonometric functions—that is, to write them as sums of single trigonometric functions. These three rules allow us to write products of sines and cosines as sums:

```
trigLinearRules = {
    Sin[x_] Cos[y_] :> Sin[x+y]/2 + Sin[x-y]/2,
    Sin[x_] Sin[y_] :> Cos[x-y]/2 - Cos[x+y]/2,
    Cos[x_] Cos[y_] :> Cos[x+y]/2 + Cos[x-y]/2
}
```

Rules for writing products of trigonometric functions as sums

All products are written as sums.

```
In[2]:= Sin[a] Cos[b] + Sin[a] Cos[a] +
        Cos[2a] Cos[3a] /. trigLinearRules
        Cos[a]   Cos[5 a]   Sin[2 a]   Sin[a - b]
Out[2]= ------ + -------- + -------- + ---------- +
          2         2          2           2
  Sin[a + b]
  ----------
      2
```

The operator `/.` applies the rules only once to each subexpression.

```
In[3]:= Cos[a] Cos[2a] Cos[3a] Cos[4a] /. trigLinearRules
          Cos[a]   Cos[3 a]
Out[3]= (------ + --------) Cos[3 a] Cos[4 a]
            2        2
```

The operator `//.` applies the rules several times, until no more rules can be applied.

```
In[4]:= Cos[a] Cos[2a] Cos[3a] Cos[4a] //. trigLinearRules
          Cos[a]   Cos[3 a]    Cos[a]   Cos[7 a]
Out[4]= (------ + --------) (------ + --------)
            2        2          2        2
```

The result is not yet in the desired form, because it still contains (implicit) products of trigonometric functions. Only after the distributive law is applied (with `Expand[]`) can the rules be applied again.

```
In[5]:= Expand[ % ] //. trigLinearRules
```

$$\text{Out[5]=}\ \frac{\text{Cos[a]}^2}{4} + \frac{\frac{\text{Cos[2 a]}}{2} + \frac{\text{Cos[4 a]}}{2}}{4} + \frac{\frac{\text{Cos[6 a]}}{2} + \frac{\text{Cos[8 a]}}{2}}{4} + \frac{\frac{\text{Cos[4 a]}}{2} + \frac{\text{Cos[10 a]}}{2}}{4}$$

Expanding the result again shows that the rules cannot be applied again.

```
In[6]:= Expand[ % ]
```

$$\text{Out[6]=}\ \frac{\text{Cos[a]}^2}{4} + \frac{\text{Cos[2 a]}}{8} + \frac{\text{Cos[4 a]}}{4} + \frac{\text{Cos[6 a]}}{8} + \frac{\text{Cos[8 a]}}{8} + \frac{\text{Cos[10 a]}}{8}$$

Because we do not know beforehand how often we have to expand out products, it is best to use a fixed-point construction that applies the simplifier function as often as necessary.

```
In[7]:= FixedPoint[
            Function[e, Expand[e //. trigLinearRules]],
            Cos[a] Cos[2a] Cos[3a] Cos[4a]
        ]
```

$$\text{Out[7]=}\ \frac{\text{Cos[a]}^2}{4} + \frac{\text{Cos[2 a]}}{8} + \frac{\text{Cos[4 a]}}{4} + \frac{\text{Cos[6 a]}}{8} + \frac{\text{Cos[8 a]}}{8} + \frac{\text{Cos[10 a]}}{8}$$

The preceding example shows that we need additional rules to simplify powers of trigonometric functions. Mathematically, `Cos[`x`]^2` is the same as `Cos[`x`] Cos[`x`]`. The square is stored differently (as `Power[Cos[`x`], 2]`), and, as a consequence, our rules do not match. The simplest way to derive the new rules is to view `Cos[`x`]^`n as `Cos[`x`] Cos[`x`] Cos[`x`]^(`n`-2)`, and then to use the previous rules to rewrite `Cos[`x`] Cos[`x`]`. We get

```
Sin[x_]^n_Integer?Positive :> (1/2 - Cos[2x]/2) Sin[x]^(n-2)
Cos[x_]^n_Integer?Positive :> (1/2 + Cos[2x]/2) Cos[x]^(n-2)
```

The restriction of the exponent `n` to a positive integer with `n_Integer?Positive` is necessary because rational or negative exponents would lead to infinite application of the rules.

As we have seen, it is necessary to multiply out intermediate results, and then to apply the rules again. We can write a function that performs these steps for us. We call it `TrigLinear[`*expr*`]`. Furthermore, we turn our small program into a package. It is shown in Listing 6.2–1.

The expression is simplified until all trigonometric functions occur only linearly.

```
In[1]:= TrigLinear[ Sin[x]^2 Cos[x]^3 ]
```

$$\text{Out[1]=}\ \frac{\text{Cos[x]}}{8} - \frac{\text{Cos[3 x]}}{16} - \frac{\text{Cos[5 x]}}{16}$$

```
BeginPackage["ProgrammingInMathematica`TrigSimplification`"]

TrigLinear::usage = "TrigLinear[e] expands products and powers of trigonometric
    functions."

Begin["`Private`"]

trigLinearRules = {
    Sin[x_] Cos[y_] :> Sin[x+y]/2 + Sin[x-y]/2,
    Sin[x_] Sin[y_] :> Cos[x-y]/2 - Cos[x+y]/2,
    Cos[x_] Cos[y_] :> Cos[x+y]/2 + Cos[x-y]/2,
    Sin[x_]^n_Integer?Positive :> (1/2 - Cos[2x]/2) Sin[x]^(n-2),
    Cos[x_]^n_Integer?Positive :> (1/2 + Cos[2x]/2) Cos[x]^(n-2) }

SetAttributes[TrigLinear, Listable]
TrigLinear[expr_] :=
    FixedPoint[ Function[e, Expand[e //. trigLinearRules]], expr ]

End[]

Protect[TrigLinear]

EndPackage[]
```

Listing 6.2–1: TrigSimplification1.m: Linearizing products of trigonometric expressions

An important application of normal forms is to decide whether two different-looking expressions describe the same function. If we integrate a function and differentiate the result, we should get back the original function. Often, however, the result will look different. Because the linear form is a normal form for trigonometric functions, we can use `TrigLinear[]` to check the result.

First, we integrate $\sin^2 x \cos^2 x$.

```
In[2]:= Integrate[ Sin[x]^2 Cos[x]^2, x ]

        4 x - Sin[4 x]
Out[2]= --------------
             32
```

Then, we differentiate the result. It looks different from the original expression.

```
In[3]:= D[ %, x ]

        4 - 4 Cos[4 x]
Out[3]= --------------
             32
```

We put the result into normal form.

```
In[4]:= TrigLinear[ % ]

        1   Cos[4 x]
Out[4]= - - --------
        8      8
```

Our original expression is also put into normal form. Now we can see immediately that the two expressions are the same.

```
In[5]:= TrigLinear[ Sin[x]^2 Cos[x]^2 ]

        1   Cos[4 x]
Out[5]= - - --------
        8      8
```

We can check the equality of two expressions by putting both of them into normal form. This procedure is usually simpler than is trying to transform one expression into the other.

Note that the functionality of our `TrigLinear[]` is now built into *Mathematica* under the name `TrigReduce[]`.

6.2.2 Simplifying Arguments of Trigonometric Functions

`TrigLinear[]` linearizes trigonometric functions, and in doing so can introduce more complicated arguments of these functions.

The simple arguments `x` and `y` are turned into more complicated ones.

```
In[1]:= TrigLinear[ Cos[x] Cos[y] Sin[x] ]

        Sin[2 x - y]   Sin[2 x + y]
Out[1]= ------------ + ------------
             4              4
```

Let us consider the reverse process: simplifying the arguments. These two formulae allow the simplification of arguments:

$$\sin(x+y) = \sin x \cos y + \cos x \sin y\,, \tag{6.2–2}$$

$$\cos(x+y) = \cos x \cos y - \sin x \sin y\,. \tag{6.2–3}$$

It is not difficult to find rules that perform these simplifications. Again, we have to think about special cases. The term $\sin(2x)$ is the same as $\sin(x+x)$, which gives us a way to deal with multiples of arguments. More generally, we write $\sin nx$ as $\sin(x+(n-1)x)$ for positive integers n. Note that we do not need any rules for negative multiples. Trigonometric functions are put into normal form by *Mathematica*, and these normal forms do not contain minus signs, as we saw at the beginning of Section 6.2. The rules and the function `TrigArgument[]`, which applies them, are in the package TrigSimplification2.m, shown in part in Listing 6.2–2. `TrigArgument[]` uses `Together[]` to simplify the result (in the same way that `Expand[]` was used in `TrigLinear[]`).

```
TrigArgument::usage = "TrigArgument[e] writes trigonometric functions
    of sums and products as products of simple trigonometric functions."
:
trigArgumentRules = {
    Sin[x_ + y_] :> Sin[x] Cos[y] + Sin[y] Cos[x],
    Cos[x_ + y_] :> Cos[x] Cos[y] - Sin[x] Sin[y],
    Sin[n_Integer?Positive x_.] :> Sin[x] Cos[(n-1)x] + Sin[(n-1)x] Cos[x],
    Cos[n_Integer?Positive x_.] :> Cos[x] Cos[(n-1)x] - Sin[x] Sin[(n-1)x]
}
:
TrigArgument[expr_] :=
    Together[ FixedPoint[ Function[e, e //. trigArgumentRules], expr ] ]
```

Listing 6.2–2: Part of TrigSimplification2.m: Simplification of arguments

In this way, we can get back the input from line 1.

```
In[2]:= TrigArgument[ % ]
Out[2]= Cos[x] Cos[y] Sin[x]
```

Here is another example. First, we expand it out.

```
In[3]:= TrigLinear[ Sin[x]^2 ]

        1   Cos[2 x]
Out[3]= - - --------
        2      2
```

Now, we try to get back the original expression. The result looks different, however.

```
In[4]:= TrigArgument[ % ]

                2         2
        1 - Cos[x]  + Sin[x]
Out[4]= ---------------------
                  2
```

To prove that it is right, we put it into normal form.

```
In[5]:= TrigLinear[ % ]

        1   Cos[2 x]
Out[5]= - - --------
        2      2
```

The preceding example shows that `TrigArgument[]` does not give normal forms. There are several possible ways to write a trigonometric expression, if we allow products of trigonometric functions. This fact is a consequence of identities such as $\sin^2 x + \cos^2 x = 1$.

Note that the functionality of our `TrigArgument[]` is built in, under the name `TrigExpand[]`.

■ 6.2.3 Performance Considerations

In our rules for `TrigArgument[]`, we replaced `Sin[n x]` by an expression involving `Sin[(n-1) x]`. The advantage of this method is that the rule is easy to derive. Repeated application of rules performs the iteration automatically. The disadvantage is the slow speed of such rules, as run-time measurements show.

This command generates a table of the time needed for the application of our rules for $\sin(nx)$, for $n = 1, 2, \ldots, 12$. The times roughly double for each successive expression.

```
In[1]:= Table[ Timing[ TrigArgument[Sin[n x]] ][[1]],
               {n, 12} ] /. Second -> 1

Out[1]= {0., 0.01, 0.04, 0.08, 0.17, 0.35, 0.68, 1.34,
  2.68, 5.35, 10.72, 21.45}
```

We can do better by expressing $\sin(nx)$ as $\sin(\frac{n}{2}x + \frac{n}{2}x)$ for even n, and as $\sin(\frac{n+1}{2}x + \frac{n-1}{2}x)$ for odd n. This method is often called *divide and conquer*. Here are the corresponding rules. They are part of TrigSimplification3.m, shown in Listing 6.2–3.

```
Sin[n_Integer?EvenQ x_.] :>
    Sin[n/2 x] Cos[n/2 x] + Sin[n/2 x] Cos[n/2 x]

Sin[n_Integer?OddQ x_.] :>
    Sin[(n+1)/2 x] Cos[(n-1)/2 x] + Sin[(n-1)/2 x] Cos[(n+1)/2 x]

Cos[n_Integer?EvenQ x_.] :>
    Cos[n/2 x] Cos[n/2 x] - Sin[n/2 x] Sin[n/2 x]

Cos[n_Integer?OddQ x_.] :>
    Cos[(n+1)/2 x] Cos[(n-1)/2 x] - Sin[(n+1)/2 x] Sin[(n-1)/2 x]
```

Listing 6.2–3: Part of TrigSimplification3.m: Another partitioning of `Sin[n x]`

Note that we dropped the condition that n be positive, because these rules happen to work also for negative n. We would not need them for negative n because the canonicalization

turns all negative arguments into positive ones (see Section 6.1). They do not lead to infinite loops in this case as the old set of rules would.

Run times turn out to be irregular. They depend on the representation of n in binary. Asymptotically, they are of the order $O(n \log n)$, much faster than the exponential growth in the previous method.

```
In[1]:= Table[ Timing[ TrigArgument[Sin[n x]] ][[1]],
               {n, 12} ] /. Second -> 1

Out[1]= {0., 0.01, 0.04, 0.04, 0.12, 0.1, 0.17, 0.1,
  0.31, 0.27, 0.43, 0.2}
```

Divide and conquer is a general method to speed up recursive computations. For trigonometric simplifications we can do even better, because there are formulae that express $\sin(nx)$ and $\cos(nx)$ directly in terms of $\sin x$ and $\cos x$. Such formulae can be found in mathematics handbooks.

$$\sin(nx) = n\cos^{n-1}x - \binom{n}{3}\cos^{n-3}x\sin^3 x + \binom{n}{5}\cos^{n-5}x\sin^5 x - \cdots \quad (6.2\text{–}4)$$

$$\cos(nx) = \cos^n x - \binom{n}{2}\cos^{n-2}x\sin^2 x + \binom{n}{4}\cos^{n-4}x\sin^4 x - \cdots . \quad (6.2\text{–}5)$$

An important aspect of *Mathematica* is that is very easy to program such formulae directly as rules. The resulting rules, as well as similar rules for powers of trigonometric functions (for expressing $\sin^n(x)$ in terms of $\sin(kx)$ and $\cos(kx)$), are part of the final version of our package TrigSimplification.m, shown in Listing 6.2–4.

The new rules lead to a linear growth of run time, which is difficult to measure on today's fast computers.

```
In[1]:= Table[ Timing[ TrigArgument[Sin[n x]] ][[1]],
               {n, 12} ] /. Second -> 1

Out[1]= {0.01, 0.01, 0.01, 0.04, 0.03, 0.04, 0.05, 0.06,
  0.07, 0.07, 0.08, 0.08}
```

```
BeginPackage["ProgrammingInMathematica`TrigSimplification`"]

TrigLinear::usage = "TrigLinear[e] expands products and powers of
    trigonometric functions."

TrigArgument::usage = "TrigArgument[e] writes trigonometric functions
    of sums and products as products of simple trigonometric functions."

Begin["`Private`"]

trigLinearRules = {
    Sin[x_] Cos[y_] :> Sin[x+y]/2 + Sin[x-y]/2,
    Sin[x_] Sin[y_] :> Cos[x-y]/2 - Cos[x+y]/2,
    Cos[x_] Cos[y_] :> Cos[x+y]/2 + Cos[x-y]/2,
    Sin[x_]^(m1_Integer?EvenQ) :>
     With[{m=Abs[m1]},
      (2^(-m+1) (Sum[(-1)^(m/2-k) Binomial[m,k] Cos[(m-2k)x], {k, 0, m/2-1}]+
                 Binomial[m,m/2]/2))^Sign[m1] ],
    Cos[x_]^(m1_Integer?EvenQ) :>
     With[{m=Abs[m1]},
      (2^(-m+1) (Sum[Binomial[m,k] Cos[(m-2k)x], {k, 0, m/2-1}] +
                 Binomial[m,m/2]/2))^Sign[m1] ],
    Sin[x_]^(m1_Integer?OddQ) :>
     With[{m=Abs[m1]},
      (2^(-m+1) Sum[(-1)^((m-1)/2-k)*
         Binomial[m,k] Sin[(m-2k)x], {k, 0, (m-1)/2}])^Sign[m1] ],
    Cos[x_]^(m1_Integer?OddQ) :>
     With[{m=Abs[m1]},
      (2^(-m+1) Sum[Binomial[m,k] Cos[(m-2k)x], {k, 0, (m-1)/2}])^Sign[m1] ]
}

trigArgumentRules = {
    Sin[x_ + y_] :> Sin[x] Cos[y] + Sin[y] Cos[x],
    Cos[x_ + y_] :> Cos[x] Cos[y] - Sin[x] Sin[y],
    Sin[n_Integer?Positive x_.] :>
      Sum[ (-1)^((i-1)/2) Binomial[n, i] Cos[x]^(n-i) Sin[x]^i, {i, 1, n, 2} ],
    Cos[n_Integer?Positive x_.] :>
      Sum[ (-1)^(i/2) Binomial[n, i] Cos[x]^(n-i) Sin[x]^i, {i, 0, n, 2} ]
}

SetAttributes[{TrigLinear, TrigArgument}, Listable]

TrigLinear[expr_] :=
    FixedPoint[ Function[e, Expand[e //. trigLinearRules]], expr ]

TrigArgument[expr_] :=
    Together[ FixedPoint[ Function[e, e //. trigArgumentRules], expr ] ]

End[]

Protect[ TrigLinear, TrigArgument ]

EndPackage[]
```

Listing 6.2–4: TrigSimplification.m: Trigonometric simplifications

■ 6.3 Globally Defined Rules

In Section 6.2 we gave an example of a rule set: a collection of rules, and functions to apply them to expressions. Thus, *Mathematica* will not use these rules on its own; rather, you have to give a command to apply them to an expression. What we did in effect was to define a trigonometric simplification function that happens to be implemented using rules instead of some of the traditional programming constructs.

Now, we want to look at another approach. We shall set things up so that *Mathematica* automatically tries to use a set of rules on all expressions it evaluates. The advantage of this approach is that you do not have to explicitly call a function to get things done. On the downside, it is almost impossible to prevent such global rules from being applied to a certain expression should you want to do that.

■ 6.3.1 Turning Rule Sets into Definitions

Let us go back to the problem of putting products of trigonometric functions in normal form. A set of rules for doing this was given in Section 6.2 (the package TrigSimplification.m). It is rather straightforward to turn these rules into global definitions. Note that any system functions for which we define rules must be unprotected first.

Next, we turn this rule set into a proper package. Even though we do not export any functions from this package, we still define a context for it. The reason is that only in this case can we read it in using `Needs["`*Context*`` ` ``"]`. We can follow the suggestions given in the skeletal package in Section 2.4. The resulting package is shown in Listing 6.3–1.

If you compare these definitions with the corresponding rules in TrigSimplification1.m (Listing 6.2–1) you will notice the extra `Expand[]` on the right side of the definitions for powers of `Sin[]` and `Cos[]`. This ensures that higher powers will be simplified fully.

Note the tags `Sin/:` and `Cos/:` on the left side of the definitions. Without them definitions would be stored with the top level operation of the left side. For the rules above this would be `Times[]` or `Power[]`. It is normally not a good idea to store rules with the basic arithmetic operations. This slows down *every* arithmetic operation performed. Rather, the rules should be stored with the operands to which they apply.

Reading in the file sets up our rules.

```
In[1]:= << ProgrammingInMathematica`TrigDefine`
```

All possible rules are applied.

```
In[2]:= Sin[a] Cos[b]

        Sin[a - b]   Sin[a + b]
Out[2]= ---------- + ----------
            2            2
```

Powers are expanded fully, but the result looks ugly.

```
In[3]:= Sin[alpha]^4

                                1   Cos[4 alpha]
                                - + ------------
        1   Cos[2 alpha]        2        2
Out[3]= - - ------------ + ----------------
        4        2                  4
```

```
BeginPackage["ProgrammingInMathematica`TrigDefine`"]

TrigDefine::usage = "TrigDefine.m defines global rules for putting
    products of trigonometric functions into normal form."

Begin["`Private`"]    (* set the private context *)

(* unprotect any system functions for which rules will be defined *)

protected = Unprotect[ Sin, Cos ]

(* linearization *)

Sin/: Sin[x_] Cos[y_] := Sin[x+y]/2 + Sin[x-y]/2
Sin/: Sin[x_] Sin[y_] := Cos[x-y]/2 - Cos[x+y]/2
Cos/: Cos[x_] Cos[y_] := Cos[x+y]/2 + Cos[x-y]/2

(* powers *)

Sin/: Sin[x_]^n_Integer?Positive := Expand[(1/2 - Cos[2x]/2) Sin[x]^(n-2)]
Cos/: Cos[x_]^n_Integer?Positive := Expand[(1/2 + Cos[2x]/2) Cos[x]^(n-2)]

Protect[ Evaluate[protected] ]     (* restore protection of system symbols *)

End[]        (* end the private context *)

EndPackage[]  (* end the package context *)
```

Listing 6.3–1: TrigDefine.m: A package for trigonometric definitions

Another expansion is necessary to get the simplest form.

```
In[4]:= Expand[ % ]

        3   Cos[2 alpha]   Cos[4 alpha]
Out[4]= - - ------------ + ------------
        8        2              8
```

This product is not expanded fully. There are still products of trigonometric functions around. However, none of the rules given matches.

```
In[5]:= Cos[alpha] Cos[beta] Sin[beta]

         Cos[alpha - beta]   Cos[alpha + beta]
Out[5]= (----------------- + -----------------) Sin[beta]
                 2                   2
```

After expanding the products the rules once more match and fully linearize the expression. It is still in a rather complicated form, however, needing another Expand[].

```
In[6]:= Expand[ % ]

        Sin[alpha]   Sin[alpha - 2 beta]
        ---------- - -------------------
            2                 2
Out[6]= -------------------------------- +
                       2

   -Sin[alpha]   Sin[alpha + 2 beta]
   ----------- + -------------------
        2                 2
   ---------------------------------
                   2
```

We can define a function SuperExpand[] as the fixed point of Expand[].

```
In[7]:= SuperExpand[ e_ ] := FixedPoint[ Expand, e ]
```

This is the form we want.

```
In[8]:= SuperExpand[ Cos[alpha] Cos[beta] Sin[beta] ]

        -Sin[alpha - 2 beta]   Sin[alpha + 2 beta]
Out[8]= -------------------- + -------------------
                 4                      4
```

The function TrigLinear[] gave us greater control over the evaluation process. There we just kept applying the rules and expanding until the expression no longer changed. This is not so easy to achieve with global definitions.

■ 6.3.2 Functions and Tensors with Symmetries

In Section 6.1.3 we discussed rules for putting expressions involving symmetric and antisymmetric functions into normal form. Such rules are most often best defined globally for the respective objects. To express that a symbolic matrix `ms` is symmetric, make the definition

```
ms[i_, j_]/; !OrderedQ[{i, j}] := ms[j, i].
```

To express that a matrix `ma` is antisymmetric, use the definitions

```
ma[i_, j_]/; !OrderedQ[{i, j}] := -ma[j, i]
             ma[i_, i_] := 0 .
```

A rich source of symmetries are the tensors used in differential geometry and in general relativity. Let us look at a few examples of tensors with symmetries.

The totally antisymmetric tensor $\varepsilon_{i_1 i_2 \ldots i_n}$ satisfies

$$\begin{aligned} \varepsilon_{i_1 \ldots i_{k-1} i_k \ldots i_n} &= -\varepsilon_{i_1 \ldots i_k i_{k-1} \ldots i_n}\,, && 1 < k \leq n \\ \varepsilon_{i_1 i_2 \ldots i_n} &= 1\,, && i_1 < i_2 < \ldots < i_n\,. \end{aligned} \tag{6.3–1}$$

A direct implementation of our symmetry rules leads to

```
eps1[a___, x_, y_, b___]/; !OrderedQ[{x, y}] := -eps1[a, y, x, b]
eps1[a___, x_, x_, b___] := 0
eps1[a___Integer] := 1
```

Note that we need not put any condition on the last rule. The first rule in effect sorts the arguments of `eps1`, so they will be in standard order before the third rule is even looked at. The sorting method encoded by the first rule is essentially bubble sort, a rather inefficient sorting procedure. Therefore, these rules are not recommended for tensors of high rank.

We turn on tracing to observe the rules for `eps1`.

```
In[1]:= On[eps1]
```

Here we see how the arguments, which are in reverse order, are sorted in a rather slow fashion.

```
In[2]:= eps1[d, c, b, a]
eps1::trace: eps1[d, c, b, a] --> -eps1[c, d, b, a].
eps1::trace: eps1[c, d, b, a] --> -eps1[c, b, d, a].
eps1::trace: eps1[c, b, d, a] --> -eps1[b, c, d, a].
eps1::trace: eps1[b, c, d, a] --> -eps1[b, c, a, d].
eps1::trace: eps1[b, c, a, d] --> -eps1[b, a, c, d].
eps1::trace: eps1[b, a, c, d] --> -eps1[a, b, c, d].
Out[2]= eps1[a, b, c, d]
```

The final sign after sorting the arguments is the *signature* of the permutation needed to put the arguments into standard order. Here are the better rules; they compute the signature and then sort the arguments in one step.

```
e:eps2[a___]/; !OrderedQ[{a}] := Signature[{a}] Sort[Unevaluated[e]]
eps2[a___]/; !UnsameQ[a] := 0
eps2[a___Integer] := 1
```

Note the use of `Unevaluated[e]` (see Section 5.3.4) to avoid the infinite recursion that `Sort[e]` alone would cause, as the current rule would match again. The second rule is necessary, because arguments such as `eps2[i, i]` are considered ordered, so the first rule does not match if identical arguments appear in the proper order. `UnsameQ[`e_1`, `e_2`, `…`, `e_n`]` returns `False`, if the e_i are not all pairwise distinct.

Here are the values for $\varepsilon_{\alpha\beta\gamma\delta}$ (rank 4), the *Levi–Civita tensor* in a positive orthonormal basis. (Indices range from 0 to 3 in general relativity; hence the offset 0 in `Array`.)

```
In[3]:= Array[eps2, {4, 4, 4, 4}, 0] // TableForm

Out[3]//TableForm=

0 0  0  0    0  0 0  0    0  0  0 0    0  0  0 0
0 0  0  0    0  0 0  0    0  0  0 -1   0  0  1 0
0 0  0  0    0  0 0  1    0  0  0 0    0  -1 0 0
0 0  0  0    0  0 -1 0    0  1  0 0    0  0  0 0

0 0  0  0    0  0 0  0    0  0  0 1    0  0  -1 0
0 0  0  0    0  0 0  0    0  0  0 0    0  0  0 0
0 0  0  -1   0  0 0  0    0  0  0 0    1  0  0 0
0 0  1  0    0  0 0  0    -1 0  0 0    0  0  0 0

0 0  0  0    0  0 0  -1   0  0  0 0    0  1  0 0
0 0  0  1    0  0 0  0    0  0  0 0    -1 0  0 0
0 0  0  0    0  0 0  0    0  0  0 0    0  0  0 0
0 -1 0  0    1  0 0  0    0  0  0 0    0  0  0 0

0 0  0  0    0  0 1  0    0  -1 0 0    0  0  0 0
0 0  -1 0    0  0 0  0    1  0  0 0    0  0  0 0
0 1  0  0    -1 0 0  0    0  0  0 0    0  0  0 0
0 0  0  0    0  0 0  0    0  0  0 0    0  0  0 0
```

The rules for `eps2` given above have been developed with symbolic processing in mind (where the indices are symbols, rather than integers). If the emphasis is on actual computation with integer values, a more efficient rule is

```
eps3[i___Integer] := Signature[{i}]
```

Rather than sorting integer arguments it determines the correct value in one step. Note that it works correctly even if arguments appear more than once, because `Signature[{…, `i`, …, `i`, …}]` gives 0.

■ 6.4 Pattern Matching for Rules

In Chapter 1 and Chapter 3 we have already used some of the possibilities of pattern matching. There, we defined rules for procedures such as `CartesianMap[]` or `PolarMap[]`. These rules look like `f[`*arg*$_1$`, `*arg*$_2$`, ...] := `*body*. In principle, every rule is of this form, but now we take a different point of view. We do not view a rule like `Sin[x_] Cos[y_] := ...` encountered in Section 6.3.1 as defining a procedure for `Times[]`, the top-level operation of the left side. Rather, we view it as specifying an arbitrarily complicated pattern that is to be replaced by the right side of the rule whenever it occurs.

For technical reasons, each rule has to be associated with a symbol. A definition like `f[x_] := `*body* naturally belongs to `f`, the head of the left side. If the head of the left side is an arithmetic operation, as in `Sin[x_] Cos[y_] := ...`, the rule should be associated with one of the arguments, if possible. This is done with a tag, for example, `Sin/: Sin[x_] Cos[y_] := ...`. The symbol with which the rule is to be associated must either be the head of the left side (the default) or the head of one of the arguments of the top-level operation.

When we give a formula such as

$$\sin(x + y) = \sin x \cos y + \cos x \sin y$$

we are quite good at recognizing the expression $\sin(x+y)$ even when it comes in disguise as in $\sin(2\alpha)$ or $\sin(a+b+c)$, and can apply the formula even in these cases. The corresponding definition in *Mathematica* is

```
Sin[x_ + y_] := Sin[x] Cos[y] + Sin[y] Cos[x]
```

as we saw in Section 6.2.2. In internal form, the left side of this rule is `Sin[Plus[x_, y_]]`, while the two examples given are `Sin[Times[2, alpha]]` and `Sin[Plus[a, b, c]]`, and so the rule would not match because there is no way of filling in expressions for `x_` and `y_` that make `Sin[Plus[x_, y_]]` equal to either of the examples. While we do in fact need a special rule to match cases like $\sin(2\alpha)$, we do not need special rules for the second case. In the following two subsections, we look at the two facilities that *Mathematica* provides for making the design of rules easier: attributes and defaults.

■ 6.4.1 Pattern Matching for Flat and Orderless Functions

The two arithmetic operations addition and multiplication have the attributes `Flat` and `Orderless` defined. This causes their arguments to be sorted in standard order and nested expressions to be flattened out. For example, the expression `(b + c) + a` or `Plus[Plus[b, c], a]` is first turned into `Plus[b, c, a]` and then sorted to give

Plus[a, b, c]. Pattern matching takes these attributes into account and can reverse this process. To match the expression Sin[Plus[a, b, c]] with Sin[Plus[x_, y_]], it is first turned into Sin[Plus[a, Plus[b, c]]] and then it matches with x becoming a and y becoming b+c.

This rule does not do anything particularly useful, but it shows how the pattern matches by returning the values of the two pattern variables x and y.

```
In[1]:= f[x_ + y_] := {x, y}
```

This match is as expected.

```
In[2]:= f[a + b]
Out[2]= {a, b}
```

Mathematica applies the attribute Flat in reverse to find a match. There is no guarantee as to how this will be done, as (a+b)+c or a+(b+c).

```
In[3]:= f[a + b + c]
Out[3]= {a, b + c}
```

An operation that is associative but not commutative should have the attributes Flat and OneIdentity.

Functional composition is a typical associative, but not commutative, operation.

```
In[4]:= Attributes[ Composition ]
Out[4]= {Flat, OneIdentity, Protected}
```

Here, we define our own associative operation h.

```
In[5]:= SetAttributes[h, {Flat, OneIdentity}];\
        h[x_, y_] := hp[x, y]
```

Associativity is taken into account for pattern matching and the expression is transformed into a nested application of the binary operation hp.

```
In[6]:= h[a, b, c, d]
Out[6]= hp[a, hp[b, hp[c, d]]]
```

For many associative operations, the result of applying them to a single argument returns the argument.

```
In[7]:= {Plus[a], Times[b], Composition[f]}
Out[7]= {a, b, f}
```

We can define this behavior also for our operation h. Contrary to what one might expect, this rule is not implied by the attribute OneIdentity.

```
In[8]:= h[x_] := x
```

Now, the single-argument form is simplified nicely.

```
In[9]:= h[a]
Out[9]= a
```

For many associative operations, the result of applying them to no argument gives the neutral element of the operation.

```
In[10]:= {Plus[], Times[], Composition[]}
Out[10]= {0, 1, Identity}
```

We can express that h0 is the neutral element of h.

```
In[11]:= h[] = h0;
```

To summarize, here is the template for implementing an associative operation h:

```
SetAttributes[h, {Flat, OneIdentity}] (* attributes *)
h[x_, y_] := ...  (* code for binary operation *)
h[x_] := x        (* OneIdentity *)
h[] = h0          (* neutral element *)
```

If you do not give a definition for the binary operation `h[x_, h_]` you cannot give the rule `h[x_] := x`. It would lead to infinite iteration. This is a longstanding bug of the pattern matcher. As a workaround, define the rule *before* giving the attributes.

6.4.2 Defaults for Arithmetic Operations

We have seen that we can view `Sin[a + b + c]` as an instance of `Sin[x_ + y_]`. But what about `Sin[a]`? *Mathematica* does not treat this as `Sin[a + 0]` and so the rule would not match. In fact, we would not want it to match because the right side would be `Sin[a] Cos[0] + Sin[0] Cos[a]`, which simplifies back to `Sin[a]`.

In other cases, this behavior would be quite useful. For example, we can define a rule for finding the derivatives of powers as

```
diff[x_^n_, x_] := n x^(n-1).
```

This rule does not work for the case `diff[x, x]` because the first argument is not a power. We know how to make it work in this case: we can declare the exponent optional by using `x_^n_.` instead of `x_^n_`.

This is almost the same definition as in the preceding subsection except that now the second term in the sum is optional.

```
In[1]:= f[x_ + y_.] := {x, y}
```

This expression is treated as `f[a + 0]` and the rule matches even though `Plus[]` does not occur in the expression at all!

```
In[2]:= f[ a ]
Out[2]= {a, 0}
```

Defaults are defined for addition, multiplication, and exponentiation. You can define defaults for your own functions through assignments to `Default[`*f*`]`.

The default for arguments of `p` is now set to 17.

```
In[3]:= Default[p] = 17
Out[3]= 17
```

Such an assignment is automatically stored with `p` and not with `Default`.

```
In[4]:= ?p
Global`p
Default[p] = 17
```

Here is a rule involving default arguments for `p`.

```
In[5]:= g[p[x_, y_.]] := {x, y}
```

The default is supplied and the rule matches.

```
In[6]:= g[p[5]]
Out[6]= {5, 17}
```

`p` must be present however for the rule to match.

```
In[7]:= g[a]
Out[7]= g[a]
```

To achieve the same effect as with `Plus[]` above, we need to set the attribute `OneIdentity` for our function (addition and multiplication both have this attribute set).

We shall define the same rules for `q` as we did for `p`, but we also set the attribute `OneIdentity`.

```
In[8]:= SetAttributes[q, OneIdentity]
```

The default for arguments of `q` is now set to 18.

```
In[9]:= Default[q] = 18
Out[9]= 18
In[10]:= g[q[x_, y_.]] := {x, y}
```

It matches even though `q` is not there at all.

```
In[11]:= g[a]
Out[11]= {a, 18}
```

`Plus[]` has all these attributes and defaults already defined.

```
In[12]:= ??Plus
x + y + z represents a sum of terms.

Attributes[Plus] =
  {Flat, Listable, NumericFunction, OneIdentity,
   Orderless, Protected}

Default[Plus] := 0
```

`OneIdentity` means that `f`[x] is the same as x. The expression `g[a]` can therefore be turned into `g[q[a]]`. The default value for the second argument of `q` is used to match the rule.

■ 6.4.3 Conditional Pattern Matching

There are two ways of restricting the expressions that match a pattern. You can give a predicate in the form f[x_?*pred*] directly following a pattern variable or you can put a condition on a whole pattern in the form g[x_, y_]/; *condition*. The former is used to restrict the matching of a single slot in the pattern. The second form is used for more complicated cases where the condition involves the values of several pattern variables.

> Conditions can also be given at the end of the right side of a rule, in the form g[x_, y_] := *expr* /; *condition*. Historically, this form was the only one possible. It is now discouraged in favor of g[x_, y_]/; *condition* := *expr*.

Another important case is the restriction of the matches to a certain *type* of expressions, identified by their head. To make a function f accept only integer arguments, you can use f[n_Integer] := *body*. This can be combined with a predicate, for example, restricting the argument to a positive integer with

f[n_Integer?Positive] := *body*.

Apart from the better performance, this is also good programming style, putting all requirements for the arguments of the function in one place, and is preferable to

f[n_Integer] := *body* /; Positive[n].

If no built-in predicate exists, you can either define one or use a pure function. In this case, it might be better to use a condition. To accept only integers greater than 3, we would write `f[n_Integer?(#>3&)] :=` *body* or `f[n_Integer /; n > 3] :=` *body*. Note that the whole pure function has to be put in parentheses because the priority of `?` is higher than the priority of `&`.

■ 6.4.4 Subtraction and Division

All subtractions are transformed to an addition and multiplication by -1. The expression *a* `-` *b* is parsed as *a* `+ -1` *b*. This normally *prints* again as *a* `-` *b*; therefore, this transformation is not quite obvious. A division of the form *a*/*b* is transformed into *a* *b*`^-1` in a similar way.

This is the internal form of a subtraction.

```
In[1]:= HoldForm[FullForm[ a - b ]]
Out[1]= Plus[a, Times[-1, b]]
```

A division is transformed into this form.

```
In[2]:= HoldForm[FullForm[ a/b ]]
Out[2]= Times[a, Power[b, -1]]
```

It is important to keep this in mind when writing patterns for subtractions and divisions. In Section 5.5.5, we have seen that the pattern `f[x_ - y_]` on the left side of a definition is turned into `f[x_ + -1 y_]` before the rule is defined. It will therefore match the expression `f[a - b]` as intended. It will, however, not match `f[a - 2b]` or `f[a - 3]`, for example. The internal form of these two expressions are `f[a + -2 b]` and `f[a + -3]`. A negative number -2 is not stored as `-1*2` but rather as a single object. The pattern `f[x_ + -1 y_]` does not match in this case.

Because any evaluated product contains at most one negative number we can use the pattern `f[x_ + n_?Negative y_.]` to match any difference. Note that we make `y_.` optional for the case `f[a + -3]`.

Here are some expressions to test our patterns. All should match a difference, except the last one.

```
In[1]:= e = {a - b, a - 2b, a - 3, -a + b, a -b/3, a + b}

                                                 b
Out[1]= {a - b, a - 2 b, -3 + a, -a + b, a - -, a + b}
                                                 3
```

The naive pattern matches only symbolic subtractions.

```
In[2]:= Cases[ e, x_ - y_ ]
Out[2]= {a - b, -a + b}
```

Allowing for any negative number matches all cases except for a negative number alone.

```
In[3]:= Cases[ e, x_ + n_?Negative y_ ]

                                   b
Out[3]= {a - b, a - 2 b, -a + b, a - -}
                                   3
```

This matches all cases we want.

```
In[4]:= Cases[ e, x_ + n_?Negative y_. ]

                                           b
Out[4]= {a - b, a - 2 b, -3 + a, -a + b, a - -}
                                           3
```

For division we have to deal with a negative exponent instead of a negative factor. The pattern `f[x_/y_]` is turned into `f[x_ y_^-1]` before the rule is defined. It will therefore match `f[a/b]` as intended. It will, however, not match `f[a/b^2]`, for example. The internal form of this expression is `f[a b^-2]` and the pattern `f[x_ y_^-1]` does not match. Instead we should use the pattern `f[x_. y_^n_?Negative]` to match any quotient. Note that we make `x_.` optional for the case `f[1/b]` which is stored as `f[b^-1]`.

Here are some expressions to test our patterns. All should match a quotient, except the last one.

```
In[1]:= e = {a/b, a/b^2, a/Sqrt[b], 1/b, a b}

            a   a       a       1
Out[1]= {-, --, -------, -, a b}
            b    2  Sqrt[b]  b
                b
```

The naive pattern matches only symbolic divisions.

```
In[2]:= Cases[ e, x_/y_ ]

            a
Out[2]= {-}
            b
```

Allowing for any negative exponent matches all cases except for a reciprocal.

```
In[3]:= Cases[ e, x_ y_^n_?Negative ]

            a   a       a
Out[3]= {-, --, -------}
            b    2  Sqrt[b]
                b
```

This matches all cases we want.

```
In[4]:= Cases[ e, x_. y_^n_?Negative ]

            a   a       a       1
Out[4]= {-, --, -------, -}
            b    2  Sqrt[b]  b
                b
```

`x_ + n_?Negative y_.`	matching any difference
`{x, -n y}`	referring to the two terms
`x_. y_^n_?Negative`	matching any quotient
`{x, y^-n}`	referring to the two terms

Patterns for subtractions and divisions

6.4.5 Rational and Complex Numbers

The pattern `x_. + I y_.` does not match a complex number. Although a complex number prints as a sum x + I y, it is a single object in *Mathematica*. To match a complex number use `c_Complex` and refer to its real part with `Re[c]` and to its imaginary part with `Im[c]`.

The pattern `x_. y_^n_?Negative` does not match a rational number. Although a rational number prints as a fraction p/q, it is a single object in *Mathematica*. To match a rational number use `r_Rational` and refer to its numerator with `Numerator[r]` and to its denominator with `Denominator[r]`.

`r_Rational`	matching a rational number
`Numerator[r], Denominator[r]`	referring to the two terms
`c_Complex`	matching a complex number
`Re[c], Im[c]`	referring to the two terms

Patterns for rational and complex numbers

6.4.6 Performance Considerations

Rules for associative and commutative (`Flat` and `Orderless`) operations need to be designed carefully with performance in mind. Let us discuss these issues with an example: the design of a linear function. A function f is linear if it satisfies these two conditions:

$$\begin{aligned} f(cx) &= cf(x), \quad \text{for constants } c \\ f(a+b) &= f(a)+f(b). \end{aligned}$$

These two rules are easily coded in *Mathematica*:

```
constantQ[c_Symbol] := MemberQ[Attributes[c], Constant]
constantQ[c_?NumericQ] := True

f[c_?constantQ e_] := c f[e]

f[a_ + b_] := f[a] + f[b]
```

The auxiliary predicate `constantQ[]` implements our notion of a constant. In this general case, we treat constant symbols and numeric quantities as constant.

All constant factors are taken out.

```
In[1]:= f[Pi + Sqrt[2] x + 3 a x^2]

                                    2
Out[1]= f[Pi] + Sqrt[2] f[x] + 3 f[a x ]
```

If the variable on which the function f depends is given explicitly (as a second argument), an alternative definition of linearity is

```
g[c_ e_, x_]/; FreeQ[c, x] := c g[e, x]

g[a_ + b_, x_] := g[a, x] + g[b, x]
```

All terms not depending on `x` are assumed constant.

```
In[2]:= g[Pi + Sqrt[2] x + 3 a x^2, x]

                                                2
Out[2]= g[Pi, x] + Sqrt[2] g[x, x] + 3 a g[x , x]
```

The simple rule `f[a_ + b_] := f[a] + f[b]` is inefficient for long sums, because it removes only one summand at a time; it may also hit the recursion limit `$RecursionLimit`. A long sum, such as f[e_1 + e_2 + ... + e_n] is eventually turned into f[e_1] + f[e_2] + ... + f[e_n]. That is, `f` is *threaded* over the sum.

We can use `Thread[]` to perform the desired operation in one step.

```
In[3]:= Thread[h[e1+e2+e3+e4], Plus]
Out[3]= h[e1] + h[e2] + h[e3] + h[e4]
```

A more efficient implementation of linearity, therefore, is this rule:

```
a:h[_Plus] := Thread[Unevaluated[a], Plus]
```

`Unevaluated[]` is used to avoid the infinite recursion that would have happened if `a` were evaluated as argument of `Thread[]` (see also Section 6.3.2).

6.5 Traversing Expressions

In this section we look at functions that interact with the syntax of *Mathematica*'s expressions. We are concerned only with the *internal* form of expressions, the one into which all input is first translated (by the so-called *parser*).

6.5.1 The Syntax of Expressions

A computer language needs a precise *syntax*, a set of rules that describe all formally correct expressions. You as a user of a programming language need to know how to write down your input and the parser for the language must be able to uniquely recognize your input.

A language is usually defined by two concepts. The first notion is that of the building blocks of all expressions, the things that cannot be taken apart further, sometimes called *atoms*. In *Mathematica* these atoms are the symbols, numbers, and strings.

The second concept gives the rules for making more complicated expressions from simpler ones. In *Mathematica* there is only one such rule. Given expressions e_0, e_1, ..., e_n, for $n \geq 0$, the following is also an expression:

$$e_0\texttt{[}e_1\texttt{, ..., }e_n\texttt{]}.$$

If $n = 0$, this expression looks like e_0`[ ]`. All expressions are built up in this way by starting with some atoms and using the above rule many times. e_0 is called the *head* of the expression and the $e_1, \ldots, e_n$ are called the *elements*.

For example, let us try to understand how the expression

```
Derivative[1][f][x]
```

is constructed. In this case n is 1 and e_0 is the expression `Derivative[1][f]` and e_1 is the expression `x`. `x` is a symbol and we have reached an atom. `Derivative[1][f]` is again built up according to our rule, with $n = 1$, e_0 being `Derivative[1]` and e_1 being the symbol `f`. We need to apply the rule one more time to `Derivative[1]`. e_0 is the symbol `Derivative` and e_1 is the number `1`. We have now completely decomposed the expression into its building blocks. This decomposition is unique. There is no other way we could have applied the rule.

6.5.2 Defining Your Own Language

You can define your own rules for a subset of *Mathematica*'s expressions. The set of all expressions that satisfy the rules is technically called a *language*. The rules are called a *grammar*.

For an example, we define a grammar for "algebraic expressions." We have to define the atoms and the rules for combining algebraic expressions to new ones.
The atoms of algebraic expressions are:

- Integer numbers are algebraic expressions.
- Rational numbers are algebraic expressions.
- Complex numbers with integer or rational parts are algebraic expressions.
- Symbols are algebraic expressions.

Given algebraic expressions $e_1, \ldots, e_n$, the following are also algebraic expressions:

- `Plus[`e_1`, ..., `e_n`]` (the sum of algebraic expressions).
- `Times[`e_1`, ..., `e_n`]` (the product of algebraic expressions).
- `Power[`e_1`, `r`]`, where r is an integer or rational number (rational powers of algebraic expressions).

By writing expressions in their internal forms, you can convince yourself that the examples in the following table are all algebraic expressions.

`1 - x`	`Plus[1, Times[-1, x]]`
`Sqrt[a + 1]`	`Power[Plus[a, 1], 1/2]`
`x/y`	`Times[x, Power[y, -1]]`
`I`	`Complex[0, 1]`

Examples of algebraic expressions

6.5.3 Recognizing a Language

Having defined a grammar for algebraic expressions, we now want to be able to *recognize* them, that is, to find out whether a given expression is an algebraic expression. For this purpose, we define a predicate `AlgExpQ[`*expr*`]` that returns `True` or `False` depending on whether *expr* is an algebraic expression. This is very easy. We can give definitions for `AlgExpQ[]` that correspond to all of the rules in our grammar.

First, the rules that define the atoms (Listing 6.5–1). We use pattern matching for the types of *Mathematica* atoms that are algebraic expressions. We give one rule for each kind of atom.

The rule for complex numbers works because the parts of a complex number are always other numbers and so the only cases that could match are the first two rules. There is a certain ambiguity in complex numbers. We could have defined them as composite expressions of the form `Complex[`*re*`, `*im*`]`. It is better to treat them as atoms, however.

```
AlgExpQ[ _Integer ]  = True
AlgExpQ[ _Rational ] = True
AlgExpQ[ c_Complex ] := AlgExpQ[Re[c]] && AlgExpQ[Im[c]]
AlgExpQ[ _Symbol ]   = True
```

Listing 6.5–1: AlgExp.m: Rules for atoms

The rules for composite algebraic expressions are by nature recursive. For a sum of algebraic expressions to be an algebraic expression, *all* of its terms must be algebraic expressions. Therefore, we use the logical *and* `&&` on the right side. It is sufficient to give rules for sums and products of *two* terms, as we have seen in Section 6.4.1.

```
AlgExpQ[ a_ + b_ ] := AlgExpQ[a] && AlgExpQ[b]
AlgExpQ[ a_ * b_ ] := AlgExpQ[a] && AlgExpQ[b]
AlgExpQ[ a_ ^ b_Integer ]  := AlgExpQ[a]
AlgExpQ[ a_ ^ b_Rational ] := AlgExpQ[a]
```

Listing 6.5–1 (cont.): Rules for composite algebraic expressions

Finally, we need a rule that matches if none of the above does and returns `False` because everything else is *not* an algebraic expression.

```
AlgExpQ[_] = False
```

Listing 6.5–1 (cont.): Catchall for algebraic expressions

The predicate should return `True` for all the examples given earlier.

```
In[2]:= AlgExpQ[1 - x]

Out[2]= True
```

We have made `AlgExpQ[]` listable to test a whole list of expressions.

```
In[3]:= AlgExpQ[{Sqrt[a + 1], I^I, Sqrt[-1]}]

Out[3]= {True, False, True}
```

Mathematica evaluates the argument of `AlgExpQ[]` in the normal way. Even though what we typed in is not an algebraic expression according to our grammar, it evaluates to -1 which certainly is.

```
In[4]:= AlgExpQ[ Exp[I Pi] ]

Out[4]= True
```

Incidentally, we should use the ideas presented in Section 6.4.6 to make the rules for sums and products more efficient. Better for long sums and products are the following definitions:

```
AlgExpQ[HoldPattern[Plus[e__]]]  := And @@ AlgExpQ /@ {e}
AlgExpQ[HoldPattern[Times[e__]]] := And @@ AlgExpQ /@ {e}
```

(`HoldPattern[]` is necessary to prevent the evaluation of `Plus[e__]` to `e__`; see Section 5.5.5.) We could even combine the two cases into the single definition

```
AlgExpQ[HoldPattern[(Plus|Times)[e__]]]  := And @@ AlgExpQ /@ {e}
```

■ 6.5.4 Splitting Atoms

We know now that atoms are not the ultimate building blocks of the universe they were believed to be when the term *atom* was used to denote the smallest parts from which a language is built up. In most programming languages, the atoms can be manipulated in some way, too. The functions that do this operate outside the language because in the language definition the atoms *are* the fundamental units. To manipulate an atom (a symbol or a number), we can convert it to a string and then we convert the string into a list of its characters. Now we can use normal *Mathematica* operations to manipulate this list and eventually convert it back to an atom.

These concepts are taken from the programming language LISP, with which *Mathematica* shares many common concepts. The function `Explode[`*symbol*`]` turns a symbol into a list of characters that make up its name and `Intern[`*charlist*`]` is its inverse, converting a list of characters back into a symbol. The code of Atoms.m is shown in Listing 6.5–2.

```
BeginPackage["ProgrammingInMathematica`Atoms`"]

Explode::usage = "Explode[expr] turns an expression into
    a list of characters that make up its name."

Intern::usage = "Intern[charlist] turns a list of characters into an expression."

Begin["`Private`"]

Explode[ atom_ ] := Characters[ ToString[InputForm[atom]] ]

Intern[l:{_String..}] := ToExpression[ StringJoin @@ l ]

Protect[Explode, Intern]

End[]

EndPackage[]
```

Listing 6.5–2: Atoms.m: Converting expressions to lists of characters

Incidentally, these functions work for any expression, not just atoms. The pattern in the definition of `Intern[]` matches any list whose elements are all strings. There is a short section on repeated patterns of the form *pattern*`..` in Subsection 2.3.11 of the *Mathematica* book.

We get a list of the characters of the name `Explode`.

```
In[2]:= Explode[ Explode ]

Out[2]= {E, x, p, l, o, d, e}
```

We need to look at the input form of it to see that the elements of the list are indeed strings.

```
In[3]:= InputForm[ % ]

Out[3]//InputForm= {"E", "x", "p", "l", "o", "d", "e"}
```

The functions `Explode` and `Intern` are inverses. The result of applying one to the result of the other is the original expression.

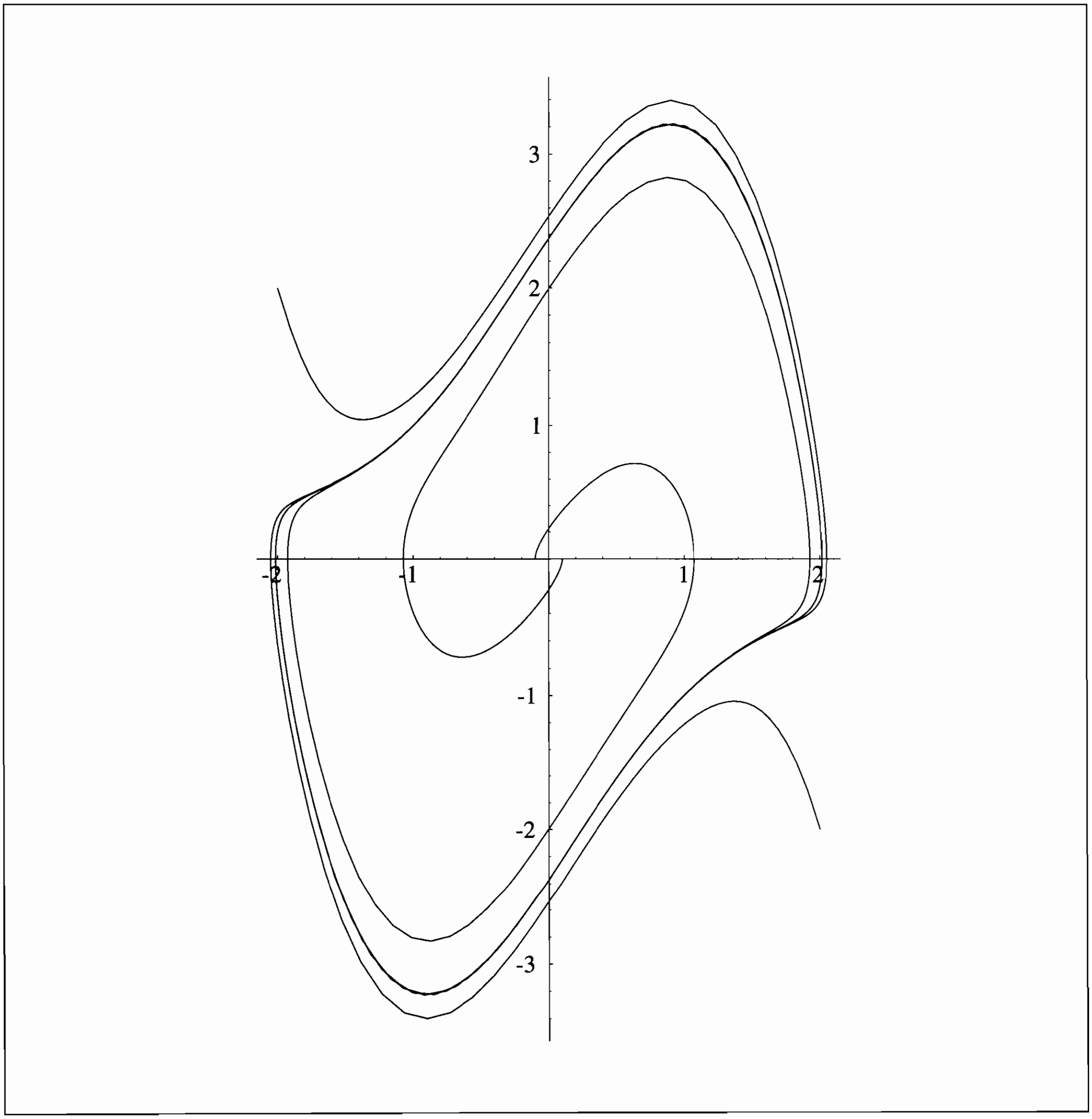
```
In[4]:= Intern[ Explode[ symbol ] ]

Out[4]= symbol
```

You can amuse yourself nesting `Explode[]`.

```
In[5]:= Explode[ Explode[ ab ] ] // InputForm

Out[5]//InputForm=
  {"{", "\"", "a", "\"", ",", " ", "\"", "b", "\"", "}"}
```

Chapter 7

Numerical Computations



Mathematica has a way of dealing with numerical computations that is not available in most other programming languages: numbers can be of any size and precision. Whenever possible, *Mathematica* maintains expressions in exact form. It does not go to a decimal approximation of `Sqrt[2]`, for example, unless you tell it to.

Section 1 is about the different kinds of numbers *Mathematica* can deal with. It explains the concepts of precision and accuracy that are important for approximate numbers. The rules for arithmetic with numbers are also explained.

In Section 2, we look at numerical evaluation (the important command `N[]`). We show how you can define your own numerical procedures.

Section 3 discusses numerical quantities, that is, expressions that denote exact numbers, such as $\sqrt{2}$ or π. The ability to perform computations with such expressions automatically is a major new feature of Version 3.

Section 4 is an application of the material treated so far and looks at numerical integration of differential equations. The package developed here can deal with simple cases and shows the principles involved. Writing a general-purpose numerical integrator is beyond the scope of this book. (Such an integrator is part of *Mathematica*.)

About the illustration overleaf:

A phase-space plot of the *Van der Pol equation* $\ddot{x} + x = \varepsilon(1 - x^2)\dot{x}$ with $\varepsilon = 1.2$, integrated numerically. Shown are four trajectories that converge rapidly toward the limit cycle. One trajectory is produced with this command:

```
Needs["ProgrammingExamples`RungeKutta`"]
eps = 1.2
RKSolve[{xdot, eps(1-x^2)xdot - x}, {x, xdot}, {0.1, 0}, {5Pi, 0.05}]
ListPlot[%, PlotJoined -> True, AspectRatio -> Automatic]
```

7.1 Numbers

There are three primitive types of numbers: integers, machine-precision floating-point numbers, and arbitrary precision floating-point numbers. Additionally, there are rational numbers and complex numbers. The real and imaginary parts of complex numbers can be any of the other number types. The numerators and denominators of rational numbers are integers. Integers and rational numbers are *exact* numbers, floating-point numbers are *approximate* numbers. You do not lose accuracy when doing computations with exact numbers. Computations with approximate numbers can only be performed to a certain accuracy and precision and a long sequence of calculations can lead to loss of accuracy.

There is no built-in limit on the size or number of digits of numbers. Ultimately, the limits will be your computer's memory size and your patience.

7.1.1 Rational and Complex Numbers

Rational numbers and complex numbers each consist of two parts. Nevertheless, they are treated as single objects for most operations, including pattern matching; see also Section 6.4.5.

A complex number is printed as `(`*a* `+` *b* `I)`. *a* is called the *real part*, *b* the *imaginary part*. Internally, this complex number is stored as `Complex[`*a*`,` *b*`]`. The symbol `I` does not occur in this representation! This is important for pattern matching. The pattern `x_ + I y_` cannot be used for matching complex numbers. Instead, use `z_Complex` and use `Re[z]` and `Im[z]` on the right side of the rule for `x` and `y`.

This is the internal form of complex numbers.

```
In[1]:= FullForm[ 2 + 3 I ]
Out[1]//FullForm= Complex[2, 3]
```

Rules like this do not match.

```
In[2]:= % /. x_ + I y_ :> {x, y}
Out[2]= 2 + 3 I
```

This is how patterns for complex numbers should be used.

```
In[3]:= %% /. z_Complex :> {Re[z], Im[z]}
Out[3]= {2, 3}
```

The symbol `I` itself is turned into a complex number.

```
In[4]:= FullForm[ I ]
Out[4]//FullForm= Complex[0, 1]
```

Again, this method of replacing imaginary parts by referring to the symbol `I` does not work in general.

```
In[5]:= f[I] + g[2I] + h[1 - I] /. I -> 0
Out[5]= f[0] + g[2 I] + h[1 - I]
```

Setting `I` equal to 0 is equivalent to taking the real part of a complex number.

```
In[6]:= f[I] + g[2I] + h[1 - I] /. z_Complex :> Re[z]
Out[6]= f[0] + g[0] + h[1]
```

Similar considerations apply to rational numbers. They are printed as *a*/*b* but are represented as `Rational[`*a*`,` *b*`]`. For pattern matching, you cannot use the pattern `a_/b_`. Use `q_Rational` and refer to `a` and `b` as `Numerator[q]` and `Denominator[q]`.

7.1.2 Approximate Numbers

Floating-point numbers are described by two quantities, their *accuracy* and their *precision*. The precision is the total number of significant digits and the accuracy describes the position of the decimal point within these digits. A number with precision p has p significant digits, in the figures below denoted by $d_1, d_2, \ldots, d_p$. *Significant digits* means that d_1 is nonzero. The accuracy is denoted by a. Let $r = p - a$. Depending on the value of a, there are three different cases:

$$
\begin{array}{lll}
(1) & 0 < a < p & \overbrace{d_1 \ldots d_r . \underbrace{d_{r+1} \cdots d_p}_{a}}^{p} \\
(2) & a \geq p & 0.\underbrace{\overbrace{0 \cdots 0}^{r}\overbrace{d_1 \cdots d_p}^{p}}_{a} \\
(3) & a \leq 0 & \overbrace{d_1 \cdots d_p}^{p}\underbrace{0 \cdots 0}_{-a}.0
\end{array}
$$

In case 1, the accuracy is less than the precision and there are digits to both sides of the decimal point. In case 2, all digits are to the right of the decimal point and there are $r = a - p$ leading zeroes (we used this formula in Section 4.4.2). In case 3, all digits are to the left of the decimal point. The accuracy is *negative!* $r = -a$ digits are undetermined and are therefore set to zero.

In all cases $r = p - a$ gives the number of digits to the left of the decimal point (r is negative in case 2) and a itself is, of course, the number of digits to the right of the decimal point (negative in case 3).

This auxiliary function reports both precision and accuracy of numbers.

```
In[1]:= pa[r_?NumberQ] := {Precision[r], Accuracy[r]}
```

This is case 1.

```
In[2]:= pa[ 5234567890.123456789 ]
Out[2]= {19, 9}
```

This is case 2.

```
In[3]:= pa[ 0.005234567890123456789 ]
Out[3]= {19, 21}
```

This is case 3.

```
In[4]:= pa[ 5.234567890123456789 10^30 ]
Out[4]= {19, -12}
```

Internally, precision and accuracy are maintained in bits and not in decimal digits. The results of `Precision[]` and `Accuracy[]` are rounded to the nearest number of decimal digits. Therefore, the calculations can be off by one digit.

This number has 19 decimal digits, but because of rounding the result of `Precision[]` is 18.

```
In[5]:= pa[ 0.1234567890123456789 ]
Out[5]= {18, 19}
```

This number also has 19 decimal digits and the rounded result of `Precision[]` agrees.

```
In[6]:= pa[ 0.9234567890123456789 ]
Out[6]= {19, 19}
```

When you enter a number such as `1.0`, you will notice that its precision is not 1 or 2 but some higher number (typically 16 or 18). *Low-precision* numbers have a minimum precision, the so-called *machine* precision. For performance reasons, *Mathematica* uses the hardware of the computer on which it is running to perform low-precision approximate computations. For these numbers it is impractical to keep track of their precision and it is assumed that it is always the same. To find the machine precision of your computer, use the constant `$MachinePrecision`. You should not put its particular value into your program as this may cause it to behave differently on other computers.

High-precision computations are performed in software and it is possible to keep track of every single bit of precision. If you need full control over precision and accuracy, you should perform your calculations with a precision higher than machine precision. Exact numbers have a precision and accuracy of `Infinity`.

The number zero is an interesting special case. Its precision is 0, because there are no digits at all. The accuracy of 0.0 on input is taken to be the negative exponent of the smallest positive machine number; therefore, it depends on the computer used. If 0.0 is the result of a computation with high-precision numbers, then its accuracy depends on the accuracy of the numbers in this computation.

This is the machine precision of the computer on which this book was formatted.

```
In[7]:= $MachinePrecision
Out[7]= 16
```

Here are precision and accuracy of the approximate number zero on input.

```
In[8]:= pa[0.0]
Out[8]= {0, 308}
```

This 0.0 is the result of a subtraction. The accuracy cannot be more than that of the operands. The harmless messages are the consequence of the sophisticated accuracy control in Version 3 that is important in advanced numerical applications. The result shows that the square root was computed to a few places more than was requested.

```
In[9]:= pa[ N[Sqrt[2], 50]^2 - 2 ]
Precision::mnprec:
   Value -8 would be inconsistent with $MinPrecision;
     bounding by $MinPrecision instead.
Accuracy::mnprec:
   Value 49 would be inconsistent with $MinPrecision;
     bounding by $MinPrecision instead.
Out[9]= {0, 57}
```

Exact zero is quite a different matter.

```
In[10]:= pa[0]
Out[10]= {Infinity, Infinity}
```

The accuracy and precision of the real and imaginary parts of complex numbers is maintained separately. `Precision[]` and `Accuracy[]` report the *minimum* of the two precisions and accuracies.

This complex number has an exact real part.

```
In[11]:= z = 1.1 I
Out[11]= 1.1 I
```

Therefore its argument can be found exactly.

```
In[12]:= Arg[z]
          Pi
Out[12]= --
          2
```

The square of it is a *real* number.

```
In[13]:= z ^ 2
Out[13]= -1.21
```

This complex number has a real part that is only approximately zero.

```
In[14]:= z = 0.0 + 1.1 I
Out[14]= 0. + 1.1 I
```

Its argument cannot be found exactly.

```
In[15]:= Arg[z]
Out[15]= 1.5708
```

The square of it remains a complex number with an imaginary part of approximately 0.

```
In[16]:= z ^ 2
Out[16]= -1.21 + 0. I
```

■ 7.1.3 Combining Numbers

When two arbitrary-precision approximate numbers are added or multiplied, the precision and accuracy of the result are determined from the precision and accuracy of the operands. They are chosen so as to guarantee that all digits in the result are correctly determined from the digits of the operands.

For multiplication, the *precision* of the result is the minimum of the precisions of the operands. The accuracy is then determined by the position of the decimal point much in the same way as with long multiplication.

The product of two numbers with precision 100 and 40, respectively, has precision 40.

```
In[1]:= pa[ N[Pi, 100] N[E, 40] ]
Out[1]= {40, 39}
```

Multiplying by a machine number, however, gives a machine number.

```
In[2]:= pa[1.0 N[Pi, 100]]
Out[2]= {16, 15}
```

For addition, the *accuracy* of the result is the minimum of the accuracies of the operands. The precision is then determined by the number of digits in the result. The technique is the same as in long addition.

The sum of two numbers with accuracy 20 and 9, respectively, has accuracy 9.

```
In[3]:= pa[0.01234567890123456789 + 1234567890.123456789]
Out[3]= {18, 9}
```

Because 1.0 is a machine number with an accuracy of less than 20, the 1 in the 20th decimal position is discarded.

```
In[4]:= 1.0 + 10.^-20
Out[4]= 1.
```

If an exact number is combined with an approximate number the same rules apply with the precision and accuracy of the exact number being infinite as we have seen.

The precision of the approximate number is preserved.

```
In[5]:= pa[ 100 N[Sqrt[2], 100] ]
Out[5]= {100, 98}
```

The accuracy of the approximate number is preserved. The precision has increased considerably.

```
In[6]:= pa[ 10^100 + 123456789.0123456789 ]
Out[6]= {110, 10}
```

The last example shows that the precision can increase if we add an approximate number and an exact number. This fact is further explained in Section 7.2.4.

There is one important exception to this rule. If a machine number is added to an exact number, the precision is never increased beyond machine precision. This is done for performance reasons.

The accuracy is not increased and we again lose the digit in the 20th decimal position.

```
In[7]:= 1 + 10.^-20
Out[7]= 1.
```

If the inexact number has a higher precision, no accuracy is lost.

```
In[8]:= 1 + N[10^-20, 20]
Out[8]= 1.000000000000000000010000000000000000000
```

■ 7.1.4 Setting Precision and Accuracy

It is possible to "cheat," that is, to artificially raise precision and accuracy of numbers. This is useful to perform intermediate computations at a higher accuracy. In most cases, the accuracy of the result should then be lowered back to what is justified by the input. We used this technique in Section 4.4.2.

This is a machine approximation of 1/3.

```
In[1]:= third = 1./3
Out[1]= 0.333333
```

The input form to base 2 shows all digits of the internal representation.

```
In[2]:= InputForm[BaseForm[%, 2]]
Out[2]//InputForm=
  2^^0.01010101010101010101010101010101010101010101010101\
0101
```

This function returns a number with the precision increased to 30 digits. The added digits are normally not 0 in base 10 representation.

```
In[3]:= SetPrecision[third, 30]
Out[3]= 0.333333333333333331482961625625
```

The added digits are 0 in base two, the internal representation for numbers. These trailing zeroes are not shown on output; instead, the exact precision (in bits) of the number is written out in the form *number`precision*. This representation makes it possible to restore the complete internal form of the number when it is read back into *Mathematica*.

```
In[4]:= InputForm[BaseForm[%, 2]]
Out[4]//InputForm=
  2^^0.01010101010101010101010101010101010101010101010101\
10101`99.6578
```

Indeed, the given precision is exactly equal to 30 decimal digits.

```
In[5]:= Log[2.0, 10^30]
Out[5]= 99.6578
```

This sets the *accuracy* to 30. Again the extra digits are not 0 in base 10.

```
In[6]:= SetAccuracy[1./9, 30]
Out[6]= 0.111111111111111110494320541875
```

■ 7.1.5 Arithmetic with Mathematical Constants

Mathematica knows about the mathematical constants `Pi`, `Degree`, `GoldenRatio`, `E`, `EulerGamma`, and `Catalan`. None of these can be represented as an exact number and they are consequently left alone when encountered in an expression. If these constants come in contact with an approximate number, however, they are converted to their numerical approximation.

π is left as an exact number in this exact expression.

```
In[1]:= 1 + Pi
Out[1]= 1 + Pi
```

Here, however, π is evaluated numerically.

```
In[2]:= 1.0 + Pi
Out[2]= 4.14159
```

To get an approximate value of a constant, simply multiply it by 1.0.

```
In[3]:= 1.0 EulerGamma
Out[3]= 0.577216
```

The numerical approximation is computed to the precision or accuracy of the inexact operand. In this way, no accuracy is lost.

```
In[4]:= N[1, 40] E
Out[4]= 2.718281828459045235360287471352662497757
```

Mathematical constants are numeric quantities; see Section 7.3.

■ 7.2 Numerical Evaluation

There are two circumstances under which *Mathematica* performs approximate numerical computations. If an expression contains approximate numbers, arithmetic involving these numbers and possibly other exact numbers is performed according to the rules we have just seen in Section 7.1. If a mathematical function receives an approximate number as argument, then its value is computed by some built-in algorithm and it returns an approximate result.

The second way to perform numerical computation is to apply the command `N[]` to an expression. This is referred to as *numerical evaluation*.

■ 7.2.1 The `N` Command

If you apply `N[]` to an expression in the form `N[`*expr*`,` *prec*`]`, where *prec* is the optional precision, which defaults to machine precision, several things happen.

- The argument *expr* is first evaluated in the normal way.
- If a numerical rule of the form `N[`h`[`e_1`, ...,` e_n`]] := ...` has been defined for the head h of *expr*, that rule is tried. If it matches, evaluation continues with the result of the rule.
- If built-in numerical code for the head h of *expr* exists, that code is called. The evaluation continues with the result returned by this built-in procedure. The working precision is increased by at most `$MaxExtraPrecision` digits in an attempt to compute the result to precision *prec*.
- If *expr* is a numeric quantity, any exact numbers in *expr* are converted to approximate numbers with up to `prec + $MaxExtraPrecision` digits in an attempt to compute the result to precision *prec*.
- For a normal expression h`[`e_1`, ...,` e_n`]`, `N[]` applies itself recursively to the head h and all elements e_i. The application to arguments can be prevented with the attributes `NHoldFirst` and `NHoldRest`.
- `N[`r`,` *prec*`]` converts the number r to an approximate number with precision at most *prec*. It does not increase the precision of a lower-precision approximate number.
- `N[`*const*`,` *prec*`]`, where *const* is one of the mathematical constants listed in Subsection 3.2.8 of the *Mathematica* book, computes a numerical approximation of *const* to precision *prec*. (See also Section 7.1.5.)
- For all other expressions, `N[`*expr*`,` *prec*`]` gives just *expr*.

The function NIntegrate[] is used to find the value of this definite integral.

```
In[1]:= N[ Integrate[Sin[Sin[x]], {x, 0, 1}] ]
Out[1]= 0.430606
```

Here, intermediate calculations with more than 40 digits are needed to find this 40-digit result. This ability to "look ahead" is a major new feature of Version 3.

```
In[2]:= N[Sin[Exp[100]], 40]
Out[2]= 0.1421981236582386377724503064786852563347
```

The naive way of first turning the arguments into approximate numbers of the required precision and then trying to compute an answer gives a much lower precision.

```
In[3]:= Sin[Exp[N[100, 40]]]
Out[3]= 0.
```

Indeed, the precision of this result is zero; the result is meaningless.

```
In[4]:= Precision[%]
Out[4]= 0
```

N applies itself to any parts of the expression, except those forbidden by the attribute NHoldFirst.

```
In[5]:= SetAttributes[f, NHoldFirst];\
        N[f[2, 3]]
Out[5]= f[2, 3.]
```

Exact numbers are converted to approximate numbers.

```
In[6]:= N[1/3, 30]
Out[6]= 0.333333333333333333333333333333
```

The precision is never increased. You can use SetPrecision[] to do this.

```
In[7]:= N[%, 100]
Out[7]= 0.333333333333333333333333333333
```

Mathematical constants are computed to the required precision.

```
In[8]:= N[Catalan, 40]
Out[8]= 0.9159655941772190150546035149323841107741
```

N[] does nothing in other cases.

```
In[9]:= N[somethingelse]
Out[9]= somethingelse
```

■ 7.2.2 Numerical Procedures

The built-in numerical procedures called by N[*func*[*args*...]] for some function *func* can be used directly. Their names are derived by preceding the name of the function by N, as in N*func*. A list of these functions is in Subsection 3.9.1 of the *Mathematica* book. Note the difference between N[*func*[...]] and N*func*[...]. The first form tries to find an exact answer and then computes the numerical approximation of that answer; the second form calls special numerical code without trying to find an exact answer first. Our example in the preceding subsection, N[Integrate[Sin[Sin[x]], {x, 0, 1}]], was carefully chosen so that there was no exact result of the integral.

What we see here is the numerical value of the exact value of this sum.

```
In[10]:= N[ Sum[1/i^2, {i, 1, Infinity}] ]
Out[10]= 1.64493
```

Here is the exact result as computed before N[] turns it into an approximate number.

```
In[11]:= Sum[ 1/i^2, {i, 1, Infinity} ]
           2
         Pi
Out[11]= ---
          6
```

Here, the numerical code is called directly to approximate the infinite sum.

```
In[12]:= NSum[ 1/i^2, {i, 1, Infinity} ]
Out[12]= 1.64493
```

Some of these numerical procedures have a nonstandard way of handling precision. In most cases it is impossible to give a result with a precision close to the precision of the input. By default, these numerical procedures use the value of the option `WorkingPrecision` as their working precision for internal calculations and try to give a result with a precision of at least ten digits less. This is explained further in Section 3.9 of the *Mathematica* book. With certain options you can ask for a higher precision than the default values would give. Often,

`N[Sum[`*expr*`,` *iterator*`],` *prec*`]`

succeeds in finding the desired result (depending on the value of `$MaxExtraPrecision`). In numerically unstable cases you can try

`NSum[`*expr*`,` *iterator*`, PrecisionGoal->`*prec*`, WorkingPrecision->`*prec+extra*`].`

The numerical sum is evaluated with a working precision high enough to give a result correct to 20 digits.

```
In[1]:= N[Sum[1/i^i, {i, 1, Infinity}], 20]
Out[1]= 1.2912859970626635404
```

There are many options that you will have to play with in difficult cases.

```
In[2]:= Options[ NSum ]
Out[2]= {AccuracyGoal -> Infinity, Compiled -> True,
  Method -> Automatic, NSumExtraTerms -> 12,
  NSumTerms -> 15, PrecisionGoal -> Automatic,
  VerifyConvergence -> True, WorkingPrecision -> 16,
  WynnDegree -> 1}
```

■ 7.2.3 Defining Your Own Numerical Procedures

To make your own numerical routine `Nf[]` behave like the built-in ones, you define a rule for `N[f[x_]]`. Such a rule is automatically stored with `f`. Because `N[]` takes an optional second argument, your rule should do so as well. The template for a numerical rule, therefore, looks like this.

```
N[f[x_], prec_:$MachinePrecision] := Nf[x, prec]
Nf[x_, prec_:$MachinePrecision] := {x, prec} (* numerical code goes here *)
```

Numerical.m: Defining a numerical rule

The first rule causes expressions of the form `N[f[`*arg*`]]` to call the numerical rule `Nf[]`, passing it the argument and the desired precision. The second rule for `Nf[]` is where the numerical computation will be performed. It can be called directly, and therefore we make the precision optional as well. In our template, we merely return the arguments in a list. `$MachinePrecision` gives the machine precision of the computer in use. This is the appropriate default for the precision in `N[]`.

The first rule is applied to turn the expression into `Nf[1/3, 20]`. `N[]` then makes all numbers approximate.

```
In[1]:= N[f[1/3], 20]
Out[1]= {0.33333333333333333333, 20.}
```

We can call Nf[] directly. Note that the numbers are not converted to approximate numbers.

```
In[2]:= Nf[5]
Out[2]= {5, 16}
```

Because numbers are not converted to approximate numbers if we call Nf[] directly, the first statement in the body of Nf[] should probably be nx = N[x, prec] or nx = N[x, prec + *extra*], where *extra* is a guess of how much extra precision for intermediate calculations is required. Thus, a typical outline for the numerical procedure Nf[] looks like this.

```
Nf[x_, prec_:$MachinePrecision] :=
    Module[{nx = N[x, prec]},
        ⋮               (* numerical code goes here *)
    ]
```

A typical numerical procedure

■ 7.2.4 Accuracy Control

As we saw in Section 7.2.3, built-in numerical methods typically provide the options PrecisionGoal, AccuracyGoal, and WorkingPrecision. These options allow fine-tuning their behavior in numerically critical applications. Let us see how we can add these options to one of our own numerical methods, the functions defined in the package Newton1.m from Section 4.4.2. The code described here is in the final package Newton.m. Let us first turn to NewtonZero[], the main function in this package. The result of NewtonZero[f, x_0] is a number x so that f[x] is close to zero. We can use the value of the option AccuracyGoal to determine how close to zero we should get. We have achieved n digits of accuracy if Abs[f[x]] < 10.0^-n. Note that this interpretation does not mean that x itself has necessarily an accuracy or precision of n. A precision goal cannot be interpreted in this way, because zero has a precision of 0. The default of AccuracyGoal is Automatic; in this case, we set the accuracy goal to the precision of the input (bounded below by machine precision, to guard against the case x_0 = 0). The default Automatic of WorkingPrecision translates to the accuracy goal plus a few extra digits (unless the accuracy goal is smaller than machine precision; in this case we do not want to perform any computations with arbitrary-precision numbers).

Having determined the working precision, we set the precision of x_0 to this value, using SetPrecision[]. There is one special case: if x_0 is 0.0, SetPrecision[0.0, *anything*] returns *exact* 0. In this case, we use SetAccuracy[] to get an inexact zero. If the accuracy of the initial value is smaller than the working precision, this command will increase it. In most iterative processes, including Newton iteration, the accuracy (and even the value) of the initial guess is unimportant. The new code of NewtonZero is shown in Listing 7.2–1.

Note that we treat the case where the accuracy goal is infinite specially. We do not need to adjust any numerical settings, but we test the result for exact zero. The only change required in NewtonFixedPoint is the treatment of the new options, in the same way that

```
Options[NewtonZero] = Options[NewtonFixedPoint] = {
    MaxIterations :> $RecursionLimit,
    AccuracyGoal -> Automatic,
    WorkingPrecision -> Automatic
}

extraPrecision = 10 (* the extra working precision *)

NewtonZero[ f_, x0_, opts___?OptionQ ] :=
    Module[{res, maxiter, accugoal, workprec, x = x0},
      {maxiter, accugoal, workprec} =
          {MaxIterations, AccuracyGoal, WorkingPrecision} /.
              Flatten[{opts}] /. Options[NewtonZero];
      With[{fp = f'},
        If[ accugoal === Automatic,
            accugoal = Max[Precision[x0], $MachinePrecision] ];
        If[ accugoal == Infinity, (* exact *)
            res = FixedPoint[(# - f[#]/fp[#])&, x, maxiter];
            If [ !TrueQ[f[res] === 0], Message[Newton::noconv, maxiter] ];
          , (* else approximate *)
            If[ workprec === Automatic, workprec = accugoal;
              If[ accugoal > $MachinePrecision, workprec += extraPrecision ];
            ];
            x = SetPrecision[x, workprec];
            If[x === 0, x = SetAccuracy[x, workprec]];
            Block[{$MaxPrecision = workprec},
              res = FixedPoint[(# - f[#]/fp[#])&, x, maxiter]
            ];
            If [ !TrueQ[Abs[f[res]] <= 10.0^-accugoal],
                 Message[Newton::noconv, maxiter] ];
        ];
        res
      ]
    ]
⋮

optnames = First /@ Options[ NewtonFixedPoint ]

NewtonFixedPoint[ f_, x0_, opts___?OptionQ ] :=
    Module[{optvals},
        optvals = optnames /. Flatten[{opts}] /. Options[NewtonFixedPoint];
        NewtonZero[ (f[#] - #)&, x0, Thread[optnames -> optvals]]
    ]
```

Listing 7.2–1: Part of Newton.m: Options for controlling precision

we treated the old option `MaxIterations` (see Section 4.4.2). Because there are now several options, we use a general code that works for any list of options.

The defaults of `Automatic` for the accuracy goal and the working precision should give us a zero of the cosine to at least 30 digits.

```
In[1]:= NewtonZero[ Cos, N[1, 30] ]

Out[1]= 1.5707963267948966192313216916397514420
```

Indeed, we get a few extra digits.

```
In[2]:= Cos[ % ]

             -42
Out[2]= 0. 10
```

Here we ask for 40 digits accuracy, but give only a low-precision initial value.

```
In[3]:= NewtonZero[ Cos, 1.0, AccuracyGoal -> 40 ]
Out[3]= 1.5707963267948966192313216916397514421
```

Nevertheless, we achieve our goal.

```
In[4]:= Cos[ % ]
               -43
Out[4]= 0. 10
```

Here is the Golden Ratio to 90 digits, found as the fixed point of $x \mapsto x + 1/x$.

```
In[5]:= NewtonFixedPoint[ 1+1/#&, 1, AccuracyGoal -> 90 ]
Out[5]= 1.61803398874989484820458683436563811772030917980\
  5762862135448622705260462818902449707207204189391
```

We can easily verify our result.

```
In[6]:= Abs[ % - GoldenRatio ]
               -107
Out[6]= 0. 10
```

Here is a case where the zero of the function is at $x = 0$. This special case is handled correctly.

```
In[7]:= NewtonZero[x(x+2), x, 1, AccuracyGoal->50]
               -61
Out[7]= 0. 10
```

Here is a case where we give an initial value of 0.0. This case, too, is handled correctly.

```
In[8]:= NewtonZero[(x+1/3)(x+2), x, 0.0]
Out[8]= -0.3333333333333
```

In a fixed-point iteration it may happen that the accuracy of the numbers increases steadily in the iteration. If this happens, the equality test used inside `FixedPoint[]` will never return true. To guard against this case, we put an upper limit on the precision for all calculations during the fixed-point iteration. We use `Block[{$MaxPrecision = workprec}, ...]` to limit the precision to our working precision.

Here you can see that an iteration of the cosine causes the precision of the numbers to increase. As a consequence, `FixedPoint[ Cos, N[1, 20] ]` would not terminate.

```
In[9]:= NestList[ Cos, N[1, 20], 10 ] // TableForm
Out[9]//TableForm= 1.0000000000000000000
                   0.5403023058681397174
                   0.8575532158463934157
                   0.6542897904977791500
                   0.79348035874256559183
                   0.70136877362275652447
                   0.76395968290065422166
                   0.72210242502670773862
                   0.75041776176376046666
                   0.731404042422509858292
                   0.744237354900556863433
```

You can prevent this runaway by bounding the maximum precision with `$MaxPrecision`.

```
In[10]:= Block[{$MaxPrecision = 40},
             FixedPoint[ Cos, N[1, 20] ]
         ]
Out[10]= 0.73908513321516064165531208767387340401
```

If you do not like the fact that `N[0]` gives 0 instead of 0.0 (you are not alone), override this behavior with the definition `N[0] = 0.0` (first you need to unprotect `Integer`).

7.3 Numeric Quantities

A major improvement in Version 3 is the better treatment of exact numeric quantities. Expressions that stand for numbers, such as `Sqrt[2]` or `Pi`, can now be treated in most circumstances in the same way as can ordinary numbers (integers, rational and complex numbers, approximate numbers).

A numeric quantity *expr* is an expression that has numerical value if `N[`*expr*`]` is evaluated, but that is not necessarily a number. The simplest examples of such numeric quantities are the mathematical constants `Pi`, `Degree`, `GoldenRatio`, `E`, `EulerGamma`, and `Catalan` (see Section 7.1.5).

The predicate `NumericQ[`*expr*`]` tells you whether an expression is a numeric quantity.

```
In[1]:= NumericQ[ GoldenRatio ]
Out[1]= True
```

Indeed, this expression does have a numerical value.

```
In[2]:= N[ GoldenRatio ]
Out[2]= 1.61803
```

Any complicated expression that has a numerical value is a numeric quantity.

```
In[3]:= NumericQ[ Sqrt[2] + Exp[Sin[Catalan]] ]
Out[3]= True
```

Numbers proper are, of course, also numeric quantities.

```
In[4]:= NumericQ[ 2 ]
Out[4]= True
```

7.3.1 Numeric Functions

A function f with the property that the function value `f[`*expr*`]` is a number whenever *expr* is an approximate number is called a *numeric function*.

A consequence is that `N[f[`*expr*`]]` is a number whenever `N[expr]` is a number; in other words, `f[`*expr*`]` is a numeric quantity whenever *expr* is. The attribute `NumericFunction` is used to specify numeric functions.

All built-in mathematical functions are numeric functions.

```
In[5]:= Attributes[ Sin ]
Out[5]= {Listable, NumericFunction, Protected}
```

The sine of an approximate number is a number.

```
In[6]:= Sin[ 2.71 ]
Out[6]= 0.418318
```

Because `E` is numeric and `Sin` is a numeric function, the expression `Sin[E]` is a numeric quantity, too.

```
In[7]:= NumericQ[ Sin[E] ]
Out[7]= True
```

There is no simple form for the exact value $\sin e$, so the expression is left alone. You need to apply `N[]` to it to get an approximate numerical value.

```
In[8]:= N[ Sin[E] ]
Out[8]= 0.410781
```

■ 7.3.2 Working with Numeric Quantities

Numeric quantities represent exact values that cannot be expressed exactly as numbers. Many calculations that can be performed with approximate numbers can also be performed with exact numeric quantities, however. The method used is to compute numerical approximations.

This inequality can be decided correctly, because the numerical values of `E^Pi` and `Pi^E` are sufficiently different.

```
In[9]:= Pi^E < E^Pi

Out[9]= True
```

Equalities between numeric quantities can usually not be decided by comparing numerical approximations.

```
In[10]:= Sqrt[2] + Sqrt[3] == Sqrt[5 + 2 Sqrt[6]]

$MaxExtraPrecision::meprec:
   $MaxExtraPrecision = 50. reached while evaluating
    Sqrt[2] + Sqrt[3] - Sqrt[5 + 2 Sqrt[6]]. Increasing
     the value of $MaxExtraPrecision may help resolve the
     uncertainty.

Out[10]= Sqrt[2] + Sqrt[3] == Sqrt[5 + 2 Sqrt[6]]
```

Algebraic methods are needed to prove that these two expression are equal.

```
In[11]:= RootReduce /@ %

Out[11]= True
```

To decide inequalities between two numeric quantities e_1 and e_2, the two quantities are turned into approximate numbers with precision at most `$MaxExtraPrecision`. If the inequality can be decided because the two numbers differ by more than their precision, the inequality can be decided for the exact numeric quantities, too.

Equalities cannot be decided in this way, however, as the preceding example showed. The fact that the numerical approximations of e_1 and e_2 are equal does not mean that e_1 and e_2 are indeed the same. (Equalities between numeric quantities and exact numbers *can* usually be decided.)

The default value of `$MaxExtraPrecision` is not sufficient to decide this inequality.

```
In[12]:= Sin[Exp[200]] > 0

$MaxExtraPrecision::meprec:
   $MaxExtraPrecision = 50. reached while evaluating
          200
    Sin[E   ]. Increasing the value of $MaxExtraPrecision
     may help resolve the uncertainty.

                 200
Out[12]= Sin[E    ] > 0
```

You can use a `Block[]` to raise the value of `$MaxExtraPrecision` for a certain calculation. This value is good enough to decide the inequality.

```
In[13]:= Block[{$MaxExtraPrecision = 100},
           Sin[Exp[200]] > 0
         ]

Out[13]= False
```

Rounding operations, too, can often be computed for numeric quantities.

A simple numerical calculation is sufficient to return this (exact) result.

```
In[14]:= Floor[ Pi ]

Out[14]= 3
```

■ 7.3.3 Example: Continued Fractions

The program described in this section can be used for both approximate and exact calculation of continued fractions. The continued-fraction expansion of a number r is the sequence of integers $a_0, a_1, a_2, \ldots$, so that (in the limit)

$$r = a_0 + \cfrac{1}{a_1 + \frac{1}{a_2 + \ldots}} . \tag{7.3–1}$$

If r is rational, the sequence is finite (that is, all $a_i = 0$ for $i > i_0$); otherwise, it is infinite. The a_i can be found as follows. Let $r_0 = r$. The first term, a_0, is equal to the integer part of r_0:

$$a_0 = \lfloor r_0 \rfloor .$$

The fractional part of r_0 is $r_0 - a_0$. Its reciprocal, $r_1 = 1/(r_0 - a_0)$, is therefore

$$r_1 = a_1 + \cfrac{1}{a_2 + \frac{1}{a_3 + \ldots}} .$$

Thus, we get

$$a_1 = \lfloor r_1 \rfloor ,$$

and so on. The function `CF[`r`, `n`]`, shown in Listing 7.3–1, computes the first $n + 1$ terms `{`a_0`, `a_1`, `a_2`, ..., `a_n`}` of the continued-fraction expansion of r. Because the floor function and the comparison with zero can be computed for numeric quantities with the methods described in Section 7.3.2, our program works for explicit numbers as well as for numeric quantities. Therefore, we use the predicate `NumericQ` in the argument declaration.

The reverse operation `CFValue[{`a_0`, `a_1`, `a_2`, ..., `a_n`}]` computes the (rational) value of a continued-fraction expansion according to Equation 7.3–1. Note the use of ∞ to start the folding operation.

Rational numbers have a finite continued-fraction expansion. Here, we have

$$\frac{3}{11} = 0 + \cfrac{1}{3 + \frac{1}{1+1/2}} .$$

```
In[1]:= CF[ 3/11, 10 ]
Out[1]= {0, 3, 1, 2}
```

Square roots have periodic continued-fraction expansions.

```
In[2]:= CF[ Sqrt[3], 12 ]
Out[2]= {1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1}
```

The value of the continued fraction is a (good) rational approximation of $\sqrt{2}$.

```
In[3]:= CFValue[ % ]
        1351
Out[3]= ----
        780
```

The approximation is correct to seven digits.

```
In[4]:= N[ Sqrt[3] - % ]
                    -7
Out[4]= -4.74482 10
```

```
BeginPackage["ProgrammingInMathematica`ContinuedFraction`"]

CF::usage = "CF[r, n] computes up to n terms of the continued fraction
    expansion of r."
CFValue::usage = "CFValue[list] gives the rational value of a continued fraction."

Begin["`Private`"]

CF[r0_?NumericQ, n_Integer?NonNegative] :=
    Module[{l = {}, r = r0, a},
        Do[ a = Floor[r];           (* integer part *)
            AppendTo[l, a];
            r = r - a;              (* fractional part; 0 <= r < 1 *)
            If[ r == 0, Break[] ];
            r = 1/r;                (* r > 1 *)
          , {n}];
        l ]

CFValue[l_List] := Fold[ 1/#1 + #2&, Infinity, Reverse[l] ]

End[]

Protect[ CF, CFValue ]

EndPackage[]
```

Listing 7.3–1: ContinuedFraction.m: Continued Fractions

Above, we performed exact arithmetic with $\sqrt{3}$. A computation starting with a machine-precision approximation of $\sqrt{3}$ leads to roundoff errors after 28 terms.

```
In[5]:= CF[N[Sqrt[3]], 30]

Out[5]= {1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1,
  2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 1, 32}
```

Nothing comes for free; the exact computation is slower than is the numerical one.

```
In[6]:= {Timing[CF[Sqrt[3], 30];],
         Timing[CF[N[Sqrt[3]], 30];]}

Out[6]= {{2.34 Second, Null}, {0.05 Second, Null}}
```

The Golden Ratio has the simplest expansion. It is defined as the solution of $x = 1 + 1/x$, from which the continued fraction can be derived immediately.

```
In[7]:= CF[GoldenRatio, 15]

Out[7]= {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

Many transcendental constants have interesting continued fractions.

```
In[8]:= CF[ Pi, 12 ]

Out[8]= {3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1}
```

This function gives the number of correct digits of the length n continued-fraction approximation of r.

```
In[9]:= CFError[r_, n_] :=
            -Log[ 10.0, Abs[r - CFValue[CF[r, n]]] ]
```

The well-known rational approximations $\frac{22}{7}$ and $\frac{355}{113}$ of π are continued-fraction values. This table shows the order of the approximation, the rational approximation, and the number of correct digits for the first four approximations.

```
In[10]:= TableForm[
           Table[{n, CFValue[CF[Pi, n]], CFError[Pi, n]},
                 {n, 4}], TableAlignments -> Center]
```

Out[10]//TableForm=

1	3	0.848959
2	$\frac{22}{7}$	2.89808
3	$\frac{333}{106}$	4.07977
4	$\frac{355}{113}$	6.57387

■ 7.3.4 Changes from Earlier Editions

Expressions such as `E < Pi` and `Floor[Log[10, 123]]` that involve numeric quantities were not evaluated in earlier version of *Mathematica*. It was necessary to wrap `N[]` around them. These uses of `N[]` can be removed from your old code.

Most of your functions that have argument restrictions of the form `f[x_?NumberQ]` or tests of the form `f[x_/;NumberQ[N[x]]]` can be recast using `f[x_?NumericQ]`.

7.4 Application: Differential Equations

Mathematica is no replacement for specialized numerical codes that are highly optimized to perform machine-precision calculations. But the merging of numerical and symbolic computation possible in *Mathematica* makes it well suited for expressing numerical algorithms, testing them out and postprocessing the results, for example, in graphical form. Its ability to carry out high-precision or exact calculations is also important. Furthermore, it contains rather efficient machine-arithmetic versions of all standard numerical methods.

In this section, we want to look at a program that combines functional programming with numerical computations for numerically integrating systems of ordinary, first-order differential equations with the *Runge–Kutta* method.

The first subsection introduces the mathematical background of the Runge–Kutta method. Its understanding is not necessary for the rest of this section, however.

7.4.1 The Runge–Kutta Method

An autonomous first-order system of differential equations is given by a vector of n functions $f_1, f_2, \ldots, f_n$ of n variables $y_1, y_2, \ldots, y_n$, and is of the following form:

$$\begin{aligned} \dot{y}_1 &= f_1(y_1, y_2, \ldots, y_n) \\ \dot{y}_2 &= f_2(y_1, y_2, \ldots, y_n) \\ &\vdots \\ \dot{y}_n &= f_n(y_1, y_2, \ldots, y_n)\,, \end{aligned}$$

where $\dot{y}$ denotes differentiation of y with respect to the independent variable t. A solution is a vector $y_1(t)$, $y_2(t)$, ..., $y_n(t)$ of n functions of t that satisfy the given equations and also an initial condition $a_1, a_2, \ldots, a_n$ at time t_0

$$\begin{aligned} y_1(t_0) &= a_1 \\ y_2(t_0) &= a_2 \\ &\vdots \\ y_n(t_0) &= a_n\,. \end{aligned}$$

An autonomous system is one in which the functions $f_1, f_2, \ldots, f_n$ do not depend explicitly on t. In this case we can take $t_0 = 0$.

Writing $\vec{y}$ for $y_1, y_2, \ldots, y_n$ and $\vec{f}$ for $f_1, f_2, \ldots, f_n$, we can write the equations and initial condition simply as

$$\begin{aligned} \dot{\vec{y}} &= \vec{f}(\vec{y}) \\ \vec{y}(t_0) &= \vec{a}\,. \end{aligned}$$

A numerical method for solving such a system finds the values of $\vec{y}$ at a number of values of t starting from the initial conditions $\vec{a}$. Given the values $\vec{y}^{(0)} = \vec{y}(t_0)$, it finds the values $\vec{y}^{(1)}$ at time $t_0 + dt$, $\vec{y}^{(2)}$ at time $t_0 + 2dt$, and so on. dt is called the *step size*.

There are many different formulae for finding the $\vec{y}^{(i)}$. They differ in the number of evaluations of the functions $\vec{f}$ that are necessary. *Higher-order* methods require many evaluations of $\vec{f}$, but they can usually find accurate solutions with a larger step size dt and, therefore, fewer steps are necessary to find $\vec{y}(t)$ for some given time t. The fourth-order Runge–Kutta formula finds $\vec{y}^{(i+1)}$, given $\vec{y}^{(i)}$, as follows.

$$\begin{aligned}
\vec{k}_1 &= dt\vec{f}(\vec{y}^{(i)}) \\
\vec{k}_2 &= dt\vec{f}(\vec{y}^{(i)} + \frac{1}{2}\vec{k}_1) \\
\vec{k}_3 &= dt\vec{f}(\vec{y}^{(i)} + \frac{1}{2}\vec{k}_2) \\
\vec{k}_4 &= dt\vec{f}(\vec{y}^{(i)} + \vec{k}_3) \\
\vec{y}^{(i+1)} &= \vec{y}^{(i)} + \frac{1}{6}\left(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4\right) .
\end{aligned}$$

7.4.2 Programming the Runge–Kutta Formula

The formula for the Runge–Kutta method can be programmed in *Mathematica* rather easily. From the beginning we want to make it as flexible as possible. We use lists for the vectors $\vec{f}$ and $\vec{y}$. We can write the code in a way that is completely independent of the number of equations n.

```
RKStep[f_, y_, y0_, dt_] :=
    Module[{ k1, k2, k3, k4 },
        k1 = dt N[ f /. Thread[y -> y0] ];
        k2 = dt N[ f /. Thread[y -> y0 + k1/2] ];
        k3 = dt N[ f /. Thread[y -> y0 + k2/2] ];
        k4 = dt N[ f /. Thread[y -> y0 + k3] ];
        y0 + (k1 + 2 k2 + 2 k3 + k4)/6
    ]
```

The code for one step with the Runge–Kutta formula

The parameters of `RKStep[]` are the list `f` of expressions describing the functions $\vec{f}$, the list `y` of the names of the variables $\vec{y}$, the list `y0` of initial conditions $\vec{y}^{(0)}$, and the step size dt.

To compute the values $\vec{f}(\vec{y}^{(0)})$ we have to substitute the elements of `y0` for the variables `y` in the expressions `f`. In *Mathematica*, this is done with a list of rules like this:

$$\texttt{f /. \{}y_1 \texttt{ -> } y_1^0\texttt{, } y_2 \texttt{ -> } y_2^0\texttt{, ..., } y_n \texttt{ -> } y_n^0\texttt{\}},$$

where y_i is the i^{th} variable and y_i^0 is the i^{th} initial value. What we are given as parameters is the list of variables and the list of initial conditions. The function `Thread[]` converts

the expression `y -> y0` or

$$\{y_1,\ y_2,\ \ldots,\ y_n\}\ \text{->}\ \{y_1^0,\ y_2^0,\ \ldots,\ y_n^0\}$$

into the desired form by interchanging the lists with the rule (see Section 4.7.1).

Nowhere do we need to know the length of these lists or the number of variables given. All the arithmetic is performed element by element, because all arithmetic functions are listable. Quite in contrast to a typical numerical code, no loops are needed, because *Mathematica*'s language is rich enough to express the underlying ideas directly.

The parameters `f`, `y`, and `dt` are constant for a given system of equations. All we need is a function of a given state `y0` that produces the next state at time `dt`. We can turn `RKStep[]` into such a function by using it as function of `y0` alone like this: `RKStep[f, y, #, dt]&`. All we have to do now is to iterate this function a suitable number of times. The top level function `RKSolve[]` does this. Listing 7.4–1 shows it together with the usual package framework. `RKSolve[]` returns a list of all the $\vec{y}$ values at times 0, dt, $2dt$, ..., $m\,dt$.

```
BeginPackage["ProgrammingInMathematica`RungeKutta`"]

RKSolve::usage =
    "RKSolve[{e1,e2,..}, {y1,y2,..}, {a1,a2,..}, {t1, dt}]
    numerically integrates the ei as functions of the yi with inital values ai.
    The integration proceeds in steps of dt from 0 to t1."

Begin["`Private`"]

RKStep[f_, y_, y0_, dt_] :=
    Module[{ k1, k2, k3, k4 },
        k1 = dt N[ f /. Thread[y -> y0] ];
        k2 = dt N[ f /. Thread[y -> y0 + k1/2] ];
        k3 = dt N[ f /. Thread[y -> y0 + k2/2] ];
        k4 = dt N[ f /. Thread[y -> y0 + k3] ];
        y0 + (k1 + 2 k2 + 2 k3 + k4)/6
    ]

RKSolve[f_List, y_List, y0_List, {t1_, dt_}] :=
    NestList[ RKStep[f, y, #, N[dt]]&, N[y0], Round[N[t1/dt]] ] /;
    Length[f] == Length[y] == Length[y0]

End[]

Protect[ RKSolve ]

EndPackage[]
```

Listing 7.4–1: Part of RungeKutta.m: Solving autonomous systems of equations

The number of integration steps is found by dividing `t1` by `dt` and rounding the result.

■ 7.4.3 Example: The Lorenz Attractor

For an example of the use of `RKSolve[]`, let us look at the equation for the Lorenz Attractor. It is a system of three equations:

$$\dot{x} \;=\; -3(x-y)$$

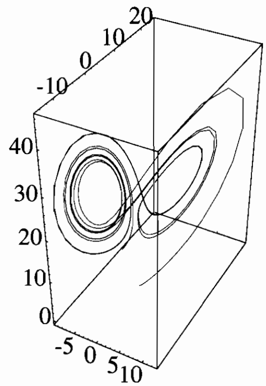
$$\begin{aligned}\dot{y} &= -xz + 26.5x - y \\ \dot{z} &= xy - z\,.\end{aligned}$$

This integrates the Lorenz equations with initial condition (0, 1, 0) from 0 to 20 in steps of size 0.04.

```
In[1]:= RKSolve[{-3(x-y), -x z + 26.5x - y, x y - z},
                   {x, y, z}, {0, 1, 0}, {20, 0.04} ];
```

The result is a list of points in space which we can plot as a line in three dimensions.

```
In[2]:= Show[ Graphics3D[{Line[%]}], Axes->Automatic ];
```



■ 7.4.4 Solving Time-Dependent Systems

In a time-dependent system, the functions $\vec{f}$ depend on t as well as on $\vec{y}$. Therefore, the equations look like

$$\begin{aligned}\dot{\vec{y}} &= \vec{f}(\vec{y}, t) \\ \vec{y}(t_0) &= \vec{a}\,.\end{aligned}$$

Rather than writing a completely new procedure for solving such equations, we can treat t as an additional dependent variable with the trivial equation $\dot{t} = 1$. We set $y_{n+1} = t$ and $f_{n+1}(y_1, y_2, \ldots, y_{n+1}) = 1$. Having done so, we use the existing code for `RKSolve[]` to solve the equations and then remove the last component from all the points in the solution before returning the solution. We do this in a second rule for `RKSolve[]`, see Listing 7.4–2.

```
RKSolve[f_List, y_List, y0_List, {t_, t0_, t1_, dt_}] :=
    Module[{res},
        res = RKSolve[Append[f, 1], Append[y, t], Append[y0, t0],
                      {t1 - t0, dt}];
        Drop[#, -1]& /@ res
    ] /; Length[f] == Length[y] == Length[y0]
```

Listing 7.4–2: RungeKutta.m (continued): Solving time-dependent systems

For an example, we look at a forced oscillation given by the equation $\ddot{x} + x = -\alpha\dot{x} + \varepsilon\cos(\omega t)$. This is one second-order equation. We can transform it into two first-order

equations with the variables $x_1 = x, x_2 = \dot{x}$. The new equations are

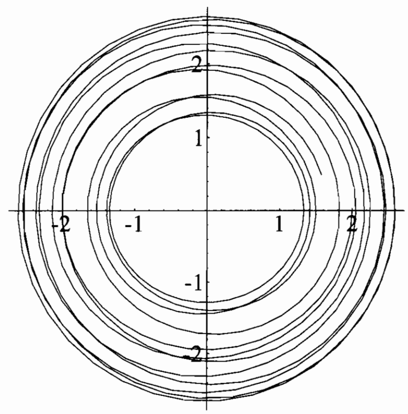
$$\begin{aligned}\dot{x}_1 &= x_2\\ \dot{x}_2 &= -x_1 - \alpha x_2 + \varepsilon\cos(\omega t)\,.\end{aligned}$$

In this example, we use $\alpha = 0.01, \varepsilon = 0.1$, and $\omega = 1.1$. The initial condition is $(2, 0)$ and time goes from 0 to 24π in steps of $\pi/18$.

```
In[1]:= RKSolve[{x2, - x1 - 0.01 x2 + 0.1 Cos[1.1 t]},
                {x1, x2}, {2, 0}, {t, 0, 24Pi, Pi/18}];
```

The result is a list of points in the plane, which we can plot as a line in two dimensions. The solution oscillates around the origin with varying amplitude.

```
In[2]:= Show[ Graphics[{Line[%]}],
              Axes->Automatic, AspectRatio->Automatic ];
```



■ 7.4.5 NDSolve

With the built-in command `NDSolve[]`, much of the material in the preceding subsections is no longer of practical value, but still serves as a useful programming example. The following calculations generate a plot of the amplitude of our oscillator from Section 7.4.4 using the built-in `NDSolve`.

This performs the numerical integration and returns a pair of interpolating functions describing the result.

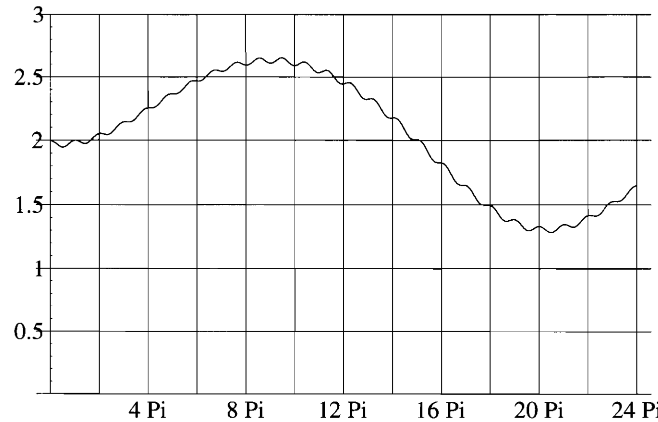
```
In[1]:= NDSolve[{x1'[t] == x2[t],
                 x2'[t] ==-x1[t] - .01 x2[t] + .1 Cos[1.1t],
                 x1[0] == 2, x2[0] == 0},
                {x1, x2}, {t, 0, 24Pi},
                 MaxSteps -> 1000 ]

Out[1]= {{x1 ->
    InterpolatingFunction[{{0., 75.3982}}, <>],
   x2 -> InterpolatingFunction[{{0., 75.3982}}, <>]}}
```

We transform the result into a point in the complex plane.

```
In[2]:= z[t_] = x1[t] + I x2[t] /. %[[1]]

Out[2]= I InterpolatingFunction[{{0., 75.3982}}, <>][t] +
  InterpolatingFunction[{{0., 75.3982}}, <>][t]
```

This plots the absolute value, or the amplitude, of the oscillation. There are both low-frequency and high-frequency components in the amplitude variations over time.

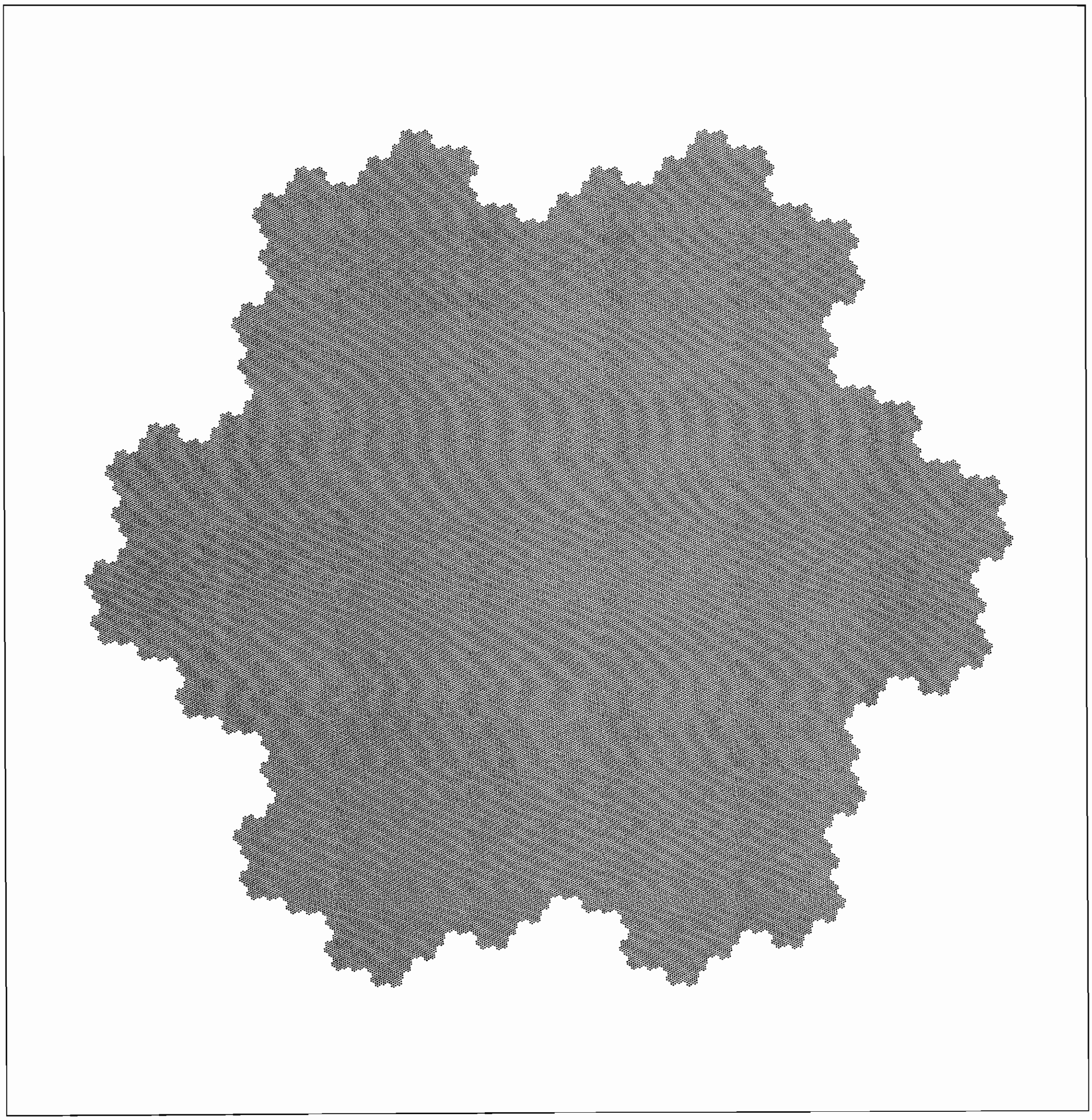
```
In[3]:= Plot[Abs[z[t]], {t, 0, 24Pi},
            PlotPoints -> 50, PlotRange -> {0, 3},
            Ticks -> {Range[0, 24Pi, 4Pi], Automatic},
            GridLines -> {Range[0, 24Pi, 2Pi], Automatic}
        ];
```



`NDSolve[]` is discussed in Subsection 3.9.7 of the *Mathematica* book.

Chapter 8

Interaction with Built-in Rules



In this chapter we look at the relationship between built-in rules and methods and user-defined ones. If you want to change the standard behavior of *Mathematica* you can give your own rules that override or augment system definitions. We show also how you can make your own functions behave like built-in ones.

In Section 1, we are concerned with the overall behavior of *Mathematica*, its main evaluation loop. Here, too, you can change the way things work by assigning functions to global variables provided for this purpose. A major application is an invisible timing command that prints evaluation timings without disturbing the computation in any way.

Section 2 looks at the two different kinds of definitions that can be made for a symbol, the so-called upvalues and downvalues, and then applies these features for defining rules for arithmetic operators and derivatives.

Adding more rules to built-in functions is the topic of Section 3. If you do not like the way things are set up, simply change them!

Section 4 is an advanced topic that deals with implementing a completely new function in *Mathematica*. The example we chose is one of the few special functions not already built-in. Programming the numerical values for a new function particularly requires advanced knowledge in numerical mathematics. However, the concepts involved in setting up the rules are easier to explain and you can skip the parts of this sections that you are not familiar with.

About the illustration overleaf:

This picture shows the sixth step in an infinite iteration to generate a fractal tile—a shape that can be used to cover the whole plane without overlapping. It has the additional property that it can be cut into seven smaller copies of itself. This image is based on an earlier version proposed by Bill Thurston. See Section 12.3.1 for more explanations.

```
Needs["IFS`"];
mids = Solve[x(x^6-1) == 0];
r7 = translation[{Re[#], Im[#]}]& /@ (x /. mids);
sr = Composition[ scale[1/Sqrt[7]], rotation[hexarotation] ];
rmaps = Composition[#, sr]& /@ r7;
hexatile = IFS[ N[rmaps] ];
pts = Flatten[ Nest[hexatile, Point[{0,0}], 6] ];
gr = Graphics[{PointSize[0.0015], pts}];
Show[gr, AspectRatio -> Automatic, PlotRange -> All];
```

8.1 Modifying the Main Evaluation Loop

Normally, *Mathematica* evaluates what you type in and then prints the value of the evaluated expression. You can modify how *Mathematica* evaluates your input or how it prints the results. This is done by assigning functions to any of the global variables `$Pre`, `$Post`, or `$PrePrint`. You can also redirect input and output to files or even programs instead of the terminal or frontend you are using.

8.1.1 Pre- and Post-Evaluation

If `$Pre` has a value, *Mathematica* evaluates the expression `$Pre[`*input*`]` where *input* is the expression that was typed in. For example, with `$Pre = Expand` *every* expression you type will be expanded.

`Expand`	expand all expressions
`Together`	put all expressions over a common denominator
`Expand[#, Trig->True]`	put trigonometric expressions into normal form
`PrintTime`	print timing for all evaluations (see Section 5.3.1)
`N`	evaluate all expressions numerically
`N[#, 100]&`	evaluate to 100 digits precision

Some functions to use for `$Pre`

If `$Post` has a value, *Mathematica* first evaluates your input in the normal way and then goes through another evaluation with the expression `$Post[`*val*`]`, where *val* is the value of the first evaluation.

`Share`	minimize storage after each evaluation
`Expand, Together,...`	same as with `$Pre` above

Some functions to use for `$Post`

If the function you use for `$Pre` or `$Post` evaluates its argument in the normal way, it makes no difference whether you assign it to `$Pre` or `$Post`, because in both cases your input is evaluated first. Assigning a function that does not evaluate its argument to `$Post` is probably useless.

If `$PrePrint` has a value, *Mathematica* first evaluates your input in the normal way and assigns the result to the next `Out[`*n*`]`. It then evaluates `$PrePrint[`*val*`]` and prints the result. A typical choice for `$PrePrint` is `Short`, which will prevent *Mathematica* from accidentally trying to print pages and pages of output if you are working with long expressions. Another common choice is `InputForm`, which will print all output in a form that is suitable to be pasted back into *Mathematica*.

`Short`	print only a one line summary
`Short[#, 5]&`	print five line summary
`InputForm`	print in input form
`FullForm`	print in internal form
`MatrixForm, ...`	print in a special format

Some functions to use for `$PrePrint`

Note that the frontend offers additional ways to control output formatting.

■ 8.1.2 Application: An Invisible Timing Command

In Section 5.3.1, we encountered the function `PrintTime[]`, which prints the time it takes to evaluate its argument and then returns the result of the evaluation. Because printing is only a side effect, It does not interfere with the normal flow of the computation in the way `Timing[]` does. In the previous subsection, we saw that we can assign `PrintTime` to `$Pre` to print the time for the evaluation of all future computations in our session.

For many applications this will be good enough. We can, however, develop a fully transparent version of `PrintTime[]` called `ShowTime[]` that has the following additional properties:

- It does not use up `$Pre`. You can still use `$Pre` for another purpose.
- Timing information can be turned on and off with the commands `On[ShowTime]` and `Off[ShowTime]`.

The key idea for reusing `$Pre` is to remember the value of `$Pre` before we assign `ShowTime` to it and to restore that value during the evaluation of the user's input. This is done by binding `$Pre` as a local variable in a block inside `ShowTime`. The code contains quite a few subtleties, and we will have a closer look at it now. The code of the package `Utilities/ShowTime.m` is shown in Listing 8.1–1.

Let us first look at how things are initially set up when `On[ShowTime]` is executed. If `ShowTime` has already been turned on, we can print an error message and do nothing. Otherwise, we assign the current value of `$Pre` to the local variable `oldPre`. This is not quite straightforward because `$Pre` need not have a value at all. Now we assign `ShowTime` to `$Pre` and remember that it has been turned on in the local variable `ison`.

```
BeginPackage["Utilities`ShowTime`"]

ShowTime::usage = "On[ShowTime] turns timing information on. Off[ShowTime]
    turns it off again.  The time taken for each command is printed
    before the result (if any)."

ShowTime::twice = "ShowTime is already on."
ShowTime::off = "ShowTime is not in effect."

Begin["`Private`"]

`oldPre          (* saved user's value of $Pre *)
ison = False     (* whether ShowTime is currently turned on *)

SetAttributes[ ShowTime, {HoldAll, SequenceHold} ]

setOldPre := If[ ValueQ[$Pre], oldPre = $Pre, Clear[oldPre] ]
setPre := If[ ValueQ[oldPre], $Pre = oldPre, Clear[$Pre] ]

ShowTime[ expr_ ] :=
    Module[{timing, result},
        Block[{$Pre},
            If[ ValueQ[oldPre],
                $Pre = oldPre;
                timing = Timing[ $Pre[expr] ]
              , (* else *)
                timing = Timing[ expr ]
            ];
            Print[ timing[[1]] ];
            setOldPre;
            If[ !ValueQ[$Pre], $Pre = Identity ]; (* this is subtle *)
            result = If[ Length[timing] == 2, timing[[2]],
                             Sequence @@ Drop[timing, 1] ]; (* restore sequences *)
        ];
        If[ !ison, setPre ];         (* turn it off, restore $Pre *)
        result
    ]

ShowTime/: On[ShowTime] := (
    If[ ison, Message[ShowTime::twice],
        setOldPre; $Pre = ShowTime; ison = True ]; )

ShowTime/: Off[ShowTime] := (
    If[ ison, ison = False,
        Message[ShowTime::off] ]; )

End[]

Protect[ ShowTime ]

EndPackage[]

On[ShowTime]
```

Listing 8.1–1: ShowTime.m: An invisible timing utility

The next time an expression *expr* is evaluated, *Mathematica* will evaluate the expression `ShowTime[`*expr*`]` because the value of `$Pre` is `ShowTime`. `ShowTime` does not evaluate its argument, and so the evaluation starts inside the body of `ShowTime`. The outer module declares a local variable that is used to hold the result of the call to `Timing[]` that is to

follow. This is similar to `PrintTime[]`. Before we call `Timing[]` on the user's input *expr*, we do two things. First, we restore the old value of `$Pre` by declaring `$Pre` as a local variable in a block and initializing it with the saved value `oldPre` (if `oldPre` does have a value). Any variable, including system symbols, can be localized. (For this purpose `Block` must be used, not `Module`; see Section 5.6.2.) Then, we apply the old value `oldPre` (if any) to *expr* before evaluating it inside `Timing[]`. In this way any value the user has defined for `$Pre` will take effect just as if `ShowTime` were not there at all.

The time it took to evaluate the user's input is now printed. Before we leave the inner block, we have to reset the value of `oldPre`, because one of the consequences of the evaluation could have been to change the user's idea of the value of `$Pre` (If the user had typed `$Pre = Expand`, for example).

To turn off `ShowTime` the user would have typed `Off[ShowTime]`. This would have happened inside `ShowTime[]`, of course. `Off[ShowTime]` simply sets the local variable `ison` to `False`, and we test for that inside `ShowTime[]`. If `ison` is `False`, we restore `$Pre` to the saved value `oldPre`. Finally, we return the result of the evaluation of the user's input.

Before reading in ShowTime.m, `$Pre` might already have some value.	`In[1]:= $Pre = Together` `Out[1]= Together`
This reads in the package and also turns `ShowTime` on.	``In[2]:= << Utilities`ShowTime` ``

From the user's point of view, `$Pre` still has the old value.	`In[3]:= $Pre`
	`0. Second`
	`Out[3]= Together`

And it works as expected.	`In[4]:= 1/x + x/(1-x)^2 + (1-x)/(1+x)`
	`0.03 Second`
	`Out[4]=` $\frac{1 - 3 x^2 + 5 x^3 - x^4}{(-1 + x)^2 x (1 + x)}$

We can remove the user's value of `$Pre` and `ShowTime` will not be affected.	`In[5]:= Clear[ $Pre ]` `0. Second`

`Together[]` is no longer applied.	`In[6]:= 1/x + x/(1-x)^2 + (1-x)/(1+x)`
	`0. Second`
	`Out[6]=` $\frac{1}{x} + \frac{x}{(1 - x)^2} + \frac{1 - x}{1 + x}$

Now we turn off `ShowTime`.	`In[7]:= Off[ShowTime]` `0. Second`
`ShowTime[]` correctly cleans up after itself. The value for `$Pre` is gone (remember that we removed it in line `In[5]`).	`In[8]:= $Pre` `Out[8]= $Pre`

■ 8.1.3 Advanced Topic: Subtleties in ShowTime

There is a line marked "subtle point" in the code of ShowTime (see Listing 8.1–1). It produces the following effect:

There is no value for `$Pre`.

```
In[9]:= $Pre
Out[9]= $Pre
```

We turn on `ShowTime`.

```
In[10]:= On[ ShowTime ]
```

Even though the user's old value of `$Pre` is normally restored, we pretend that the value `$Pre` is `Identity`, if it does not have a value at all.

```
In[11]:= $Pre
0. Second
Out[11]= Identity

In[12]:= Off[ ShowTime ]
0. Second
```

No special treatment is necessary if `ShowTime` is turned off.

```
In[13]:= $Pre
Out[13]= $Pre
```

The subtlety explained: If `$Pre` did not have a value inside `ShowTime[]`, the unevaluated symbol `$Pre` would be returned from `ShowTime[]`. As a consequence, it would be moved outside the scope of `Block[]` and there it would acquire its global value, `ShowTime`. Evaluation would proceed to turn `$Pre` into `ShowTime`. Therefore, we set `$Pre` to `Identity` before leaving the block (only if it does not have value) to bind any instances of the symbol `$Pre` that might be present in the result returned from `Timing[]`.

There is an even more subtle point: the assignment of the evaluation result contained in the variable `timing` (normally the second element of the list returned by `Timing[]`) to the variable `result`. If we did not "touch" the value in the variable `timing` before leaving the block, any instances of `$Pre` inside its value would not be reevaluated before leaving the block, and `$Pre` would again acquire its global value, instead of `Identity`. This short example shows the difference:

Here is a global variable with the value 5.

```
In[14]:= pre = 5;
```

Inside the block, `pre` loses its value and the symbol `pre` is assigned to `result`. The assignment of 7 to `pre` does not influence the value of `result` before it leaves the block. Outside the block, `result` is evaluated again and the restored value 5 of `pre` is returned.

```
In[15]:= Block[{pre},
                result = pre; pre = 7
         ]; result
Out[15]= 5
```

If we evaluate `result` before leaving the block, the new value of `pre` is used.

```
In[16]:= Block[{pre},
                result = pre; pre = 7; result = result
         ]; result
Out[16]= 7
```

The treatment of such subtleties is, of course, not the result of careful analysis, but the result of a long night spent debugging the old code.

Besides HoldAll, ShowTime is also given the attribute SequenceHold. We do this to treat inputs correctly that are of the form Sequence[*expr*$_1$, ...]. The attributes prevent any expansion of such sequences into individual arguments of ShowTime. (Timing, too, has this attribute.) If the result of the evaluation of an expression is a sequence, however, the result returned from Timing will be a list where the elements of the sequence have been spliced in. In this case, the length of the list returned from Timing will not be two, and we restore the sequence before returning.

Sequences are handled correctly on input and output.

```
In[17]:= Sequence[a, b, c]
0. Second
Out[17]= Sequence[a, b, c]
```

Here you can see that sequences that are the result of an evaluation are spliced into the list returned from Timing[].

```
In[18]:= Timing[ Sequence @@ {alpha, beta, gamma} ]
0.01 Second
Out[18]= {0. Second, alpha, beta, gamma}
```

For yet another subtlety of ShowTime, see Exercise 12.

8.2 User-Defined Rules Take Precedence

As explained in Section A.4 of the *Mathematica* book, any definitions made by the user are applied before any built-in code is executed to evaluate an expression. It is, therefore, possible to give rules that override built-in ones. Because the built-in rules will eventually be applied anyway, doing so can be a bit tricky.

8.2.1 Upvalues and Downvalues

A definition of the form

`f[`*args...*`] :=` *body*

is often called a *downvalue* for `f` because it defines a value for `f` appearing as the head of the left side of the rule. A definition of the form

`f/: g[[f[...], ...] :=` *body*

is called an *upvalue* for `f`, because `f` appears as the head of an argument of the left side. Upvalues are applied before downvalues. We used upvalues, for example, in Section 6.3.1 to give rules for products of trigonometric functions such as `Sin/: Sin[x_] Cos[y_] :=` *body*, which *Mathematica* would otherwise leave alone.

8.2.2 Definitions for Arithmetic Operations

Using upvalues for definitions involving arithmetic operations is preferred in the cases where the rules belong to a certain class of expressions, here the trigonometric expressions. If the definition is for a broader class of expressions, not characterized by a certain symbol appearing as their head, upvalues are not possible. For an example, let us develop a rule that will cause all products of sums to expand automatically. An elegant first attempt is

```
e_Times := Expand[Unevaluated[e]]
```

using `Unevaluated[]` to stop the obvious infinite recursion. This definition will, nevertheless, lead to an infinite loop because there is no guarantee that expanded expressions do not contain any more products (an example is `a b`).

The solution is to perform the expansion ourselves instead of calling `Expand[]`. The expansion of the term $(a + b)c$ is $a\ c + b\ c$, and this is all we need to expand any product:

```
(a_ + b_)c_ := a c + b c.
```

Because we shall define a rule for multiplication, we need to unprotect `Times[]`.

```
In[1]:= Unprotect[Times]
Out[1]= {Times}
```

This definition will expand all products.

```
In[2]:= (a_ + b_) c_ := a c + b c
```

It is used repeatedly until no terms with embedded addition remain.

```
In[3]:= x (u + v + w)(a - 1)
Out[3]= -(u x) + a u x - v x + a v x - w x + a w x
```

With long sums as arguments, this rule will be quite slow. We encountered the same problem with our rules for $\sin(nx)$ in Section 6.2.3. The rule will be applied many times, each time expanding only one of the terms in the sum. Following the discussion in Section 6.4.6, we could use `Thread[]` to multiply all the terms in the sum by the second argument of the product in one step:

```
e: _ _Plus := Thread[Unevaluated[e], Plus],
```

but this works only if no other term in the product is a sum. The correct operation is distribution, not threading:

```
e: _ _Plus := Distribute[Unevaluated[e]].
```

Note the pattern on the left side. Because we need it in its entirety on the right side, we give it a name, `e`. It should match a product one of whose factors must be a sum. The blanks appearing need not be named, because these parts are *not* needed on the right side. This rule will be much faster for longer sums.

The expression $a_1 + a_2 + \cdots + a_{200}$ serves as our long sum.

```
In[4]:= expr = Array[a, 200, 1, Plus];
```

It takes quite a while.

```
In[5]:= Timing[expr c;]
Out[5]= {1.34 Second, Null}
```

This deletes the old rule in preparation for the new one.

```
In[6]:= Clear[Times]
```

Here is the new rule for expansion of products.

```
In[7]:= e: _ _Plus := Distribute[Unevaluated[e]]
```

It is much faster.

```
In[8]:= Timing[expr c;]
Out[8]= {0.06 Second, Null}
```

Our rule does not expand integer powers of sums. Again we have two choices, a trivial one-step rule and a faster, more complicated formula. The trivial rule reduces exponentiation to the previous case, multiplication:

```
(a_ + b_) ^ n_Integer?Positive := (a + b)(a + b)^(n-1).
```

In this rule, we see the importance of applying user-defined rules first. The right side of the rule looks like e `e^m` which *Mathematica* would simplify to `e^(m+1)`. Our own rule for multiplying a sum with anything else is applied first, preventing another infinite loop. The faster rule uses the binomial formula $(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}$ to expand an integer power.

The rule will be defined for `Power`, which needs to be unprotected first.

```
In[9]:= Unprotect[Power]
Out[9]= {Power}
```

With the rule for expanding products still in effect, we define this rule for expanding powers.

```
In[10]:= (a_ + b_) ^ n_Integer?Positive :=
              (a + b)(a + b)^(n-1)
```

Again rather slow.

```
In[11]:= Timing[(x + y + z)^20;]
Out[11]= {9.2 Second, Null}
```

We get rid of this rule.

```
In[12]:= Clear[Power]
```

Now we replace it with the faster one.

```
In[13]:= (a_ + b_) ^ n_Integer?Positive :=
            Sum[ Binomial[n, i] a^i b^(n-i), {i, 0, n}]
```

This timing looks much better.

```
In[14]:= Timing[(x + y + z)^20;]
Out[14]= {0.65 Second, Null}
```

In Section 8.1.1 we saw a quite different way to multiply out all products: assign Expand to `$Post`.

■ 8.2.3 Derivatives

The derivative operator `Derivative[`n`]` represents the n^{th} derivative. It takes a function f as argument and returns another function—the n^{th} derivative $f^{(n)}$. Applying this function $f^{(n)}$ to an argument x gives $f^{(n)}(x)$, the n^{th} derivative at x. In *Mathematica*, this is written as `Derivative[`n`][`f`][`x`]`. If the derivative of f is known, then `Derivative[`n`][`f`]` returns a pure function representing that derivative, as we have seen in Section 5.1.3.

We can define a value for a derivative with `f' = fp` or `Derivative[1][f] = fp`. The head of the left side is `Derivative[1]`, which is not a symbol. The rule therefore cannot be stored with the head, as is normally the case. It is stored with the head of the head, the symbol `Derivative`. Such rules are called *subvalues*. It is usually better, however, to store it with the argument `f` with `f/: f' = fp`.

If we do not have a name for the derivative function, such as `fp` above, we can make a definition for `f'[x]`, for example, `f'[x_] := 1 + x^2`. The left side is `Derivative[1][f][x_]`. The symbol `f` is not an *argument* of this expression, so we cannot store the rule with `f`; it is stored with `Derivative`. `Derivative` is one of very few system symbols that are not protected, so rules like these can be given easily.

Because the derivative of tan is known, it is returned as a pure function.

```
In[1]:= Tan'

                2
Out[1]= Sec[#1]  &
```

The formal derivative of `f` is set to `g`.

```
In[2]:= f/: f' = g
Out[2]= g
```

Surprisingly, the rule is not used to simplify higher derivatives. (This behavior is new in Version 3.)

```
In[3]:= f''[x]
Out[3]= f''[x]
```

The reason the rule does not match is that the input is parsed as a higher derivative directly, not as the nested derivative ((f′)′).

```
In[4]:= FullForm[f'']
Out[4]//FullForm= Derivative[2][f]
```

In general, you should give your rules so that they work also for higher derivatives. The template is

f /: Derivative[n_Integer?Positive][*f*] := Derivative[n-1][*fp*],

where *fp* is the function giving the first derivative of *f*.

We clear the old definition for f′.

```
In[5]:= Clear[f]
```

Here is our more general rule for the first derivative of f.

```
In[6]:= Derivative[n_Integer?Positive][f] :=
            Derivative[n-1][g]
```

Now, higher derivatives are simplified nicely.

```
In[7]:= f'''
Out[7]= g''
```

For another example of defining derivatives for known functions, see Section 8.3.

8.3 Modifying System Function

This section focuses on modifying built-in behavior and adding new rules to existing functions.

8.3.1 Additional Rules for a Function

Mathematica does not always know about all the mathematical properties of its built-in functions. Often such properties are only valid for a restricted domain (for example, only for real-valued arguments) and because all variables are assumed to be complex-valued, no rules are applied. If you know that you will work only with real variables, you can add the necessary rules yourself. An example was given in Section 2.3.2: simplification of expressions involving `Re[]` and `Im[]`. In this section we want to look at another case, `Abs[]` and `Sign[]`.

For nonnumerical arguments x, *Mathematica* does not simplify `Abs[`x`]`. Because the absolute value $|x|$ is equal to x for positive x and equal to $-x$ for negative x, we can use the predicates `Positive[`x`]` and `Negative[`x`]` to give conditional rules for this type of simplification.

```
Abs[x_?Positive] :=  x
Abs[x_?Negative] := -x
```

Abs.m (excerpt): Simplifying absolute values

Next, we want to teach *Mathematica* about integrals and derivatives of the absolute value and sign functions. The derivative of $|x|$ is simply $\operatorname{sgn} x$ and the derivative of the sign function is 0. At $x = 0$ the sign function is not differentiable; therefore, we have to make the rules conditional. A built-in rule simplifies `Sign[0]` to `0`, which is inconsistent with our definition for the derivative of the absolute value. We override it by setting `Sign[0]` to `Indeterminate`.

```
Abs/: Derivative[n_Integer?Positive][Abs] := Derivative[n-1][Sign]

Derivative[n_Integer?Positive][Sign][x_]/; x != 0 := 0
Derivative[n_Integer?Positive][Sign][0] := Indeterminate

Sign[0] = Indeterminate (* consistency *)
```

Abs.m (excerpt): Derivatives

The integral of $\operatorname{sgn} x$ is, of course, $|x|$, and the integral of $|x|$ can be written as either $\frac{1}{2}x|x|$ or $\frac{1}{2}x^2 \operatorname{sgn} x$. The conversion between the two is given by $x \operatorname{sgn} x = |x|$.

```
Sign/: x_ Sign[x_] := Abs[x]

Abs /: Integrate[Abs[x_], x_]  := x Abs[x]/2
Sign/: Integrate[Sign[x_], x_] := Abs[x]
```

Abs.m (excerpt): Integrals

Integrating sgn z twice exercises our rules.

```
In[1]:= Integrate[ Integrate[Sign[z], z], z ]

        z Abs[z]
Out[1]= --------
           2
```

The second derivative simplifies back to the original expression.

```
In[2]:= D[ %, {z, 2} ]
Out[2]= Sign[z]
```

The first derivative of sgn x is left in a symbolic form because it is not yet known, whether x is zero or not.

```
In[3]:= Sign'[x]
Out[3]= Sign'[x]
```

The first derivative of $|x|$ at 0 is indeterminate.

```
In[4]:= Abs'[0]
Out[4]= Indeterminate
```

■ 8.3.2 Advanced Topic: Overriding System Functions

A question that appeared from time to time in electronic *Mathematica* forums is about ways to disable built-in functions. It is straightforward to add your own rules to any built-in function. As we saw, these rules will be used first. The tricky part is how to invoke the built-in code inside your rule without triggering your own rule again.

Here is the template for controlling a hypothetical built-in operation `BuiltIn`:

```
protected = Unprotect[BuiltIn]

$BuiltInActive = True

BuiltIn[...]/; $BuiltInActive :=
  Block[{$BuiltInActive = False},
    ⋮
    r = BuiltIn[...]; (* call built-in code *)
    ⋮
  ]

Protect[Evaluate[protected]]
```

For each function to override, a global variable is used to control whether the additional user-defined rule should be used. Inside this rule, `Block[]` sets the global variable to `False`. Therefore, any call to `BuiltIn[]` inside this rule will not trigger the rule again, but will use whatever other rules or built-in code are there.

There are other ways to modify system functions. Here are two more paradigms for overriding built-in behavior, together with examples for their use (taken from the standard packages).

- Define a new syntax and give rules for the cases handled by your code, but not by built-in code.

  Example: Our package Graphics‘ParametricPlot3D‘ allows iterators to be given in the form `{u, u0, u1, du}`. The built-in code accepts only iterators of the form `{u, u0, u1}`. See Section 10.1.1.

- Define a new option for a system function and trigger your rules only if this option is present in a function call.

  Example: The standard package Graphics‘Legend‘ adds the option `PlotLegend` to `Plot[]`. The definitions use a pattern of the form

  `Plot[..., PlotLegend->`*val*`, ...] := ....`

  Before the subsequent call to the built-in version of `Plot`, the additional option is filtered out.

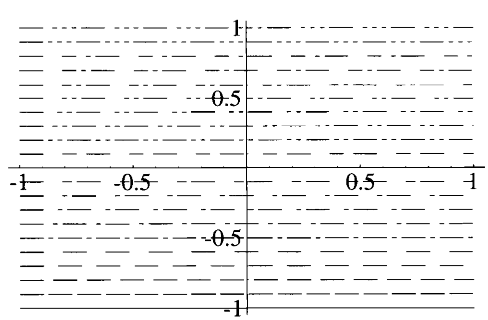
8.3.3 Example: Plotting Several Functions

For an application example let us add code to the `Plot[]` command to choose automatically different plot styles if several functions are to be plotted. By default, all curves drawn for `Plot[{`f_1`, ..., `f_n`}, `*range*`]` are drawn solid and are therefore indistinguishable. Let us choose different dashings to make it easier to tell the curves apart. The code is shown in Listing 8.3–1. The package does not export any symbols. It follows the style for such packages established in Section 2.3.2.

Note the usage message: `Plot`, of course, already has a usage message, which we should preserve. We merely add a sentence describing our addition to the code. The constant `firstStyle` represents a solid line to use for the first curve. The function `nextStyle[`*dashing*`]` produces the logically next dashing in such a way that all dashings are different. Using `NestList[]`, we can then simply produce the required number of dashing directives to use in the option setting for `PlotStyle`.

Here are 21 lines showing the different dashing patterns used.

```
In[1]:= Plot[ Evaluate[Table[a, {a, -1, 1, 0.1}]],
              {x, -1, 1} ];
```



```
BeginPackage["ProgrammingInMathematica`Plot`"]

Plot::usage = Plot::usage <> " If several functions are plotted, different
    plot styles are chosen automatically."

Begin["`Private`"]

protected = Unprotect[Plot]

$PlotActive = True

Plot[f_List, args___]/; $PlotActive :=
    Block[{$PlotActive = False},
      With[{styles = NestList[nextStyle, firstStyle, Length[Unevaluated[f]]-1]},
        Plot[f, args, PlotStyle -> styles]
      ]
    ]

(* style definitions *)

unit = 1/100
max = 5

firstStyle = Dashing[{}]

nextStyle[Dashing[{alpha___, x_, y_, omega___}]] /; x > y + unit :=
    Dashing[{alpha, x, y + unit, omega}]
nextStyle[Dashing[l_List]] :=
    Dashing[Prepend[Table[unit, {Length[l] + 1}], max unit]]

Protect[ Evaluate[protected] ]

End[]

EndPackage[]
```

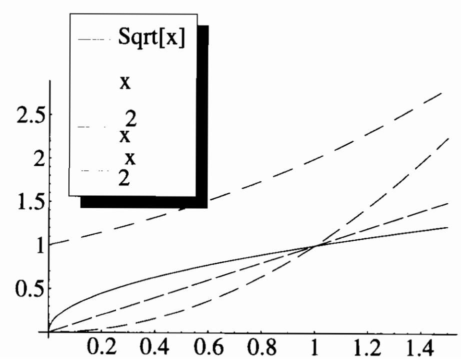
Listing 8.3–1: Plot.m: Additional definitions for plotting

We can combine our plot extension with another one, from this standard package.

```
In[2]:= Needs["Graphics`Legend`"]
```

With no effort, we can produce annotated plots of several functions.

```
In[3]:= Plot[{Sqrt[x], x, x^2, 2^x}, {x, 0, 1.5},
            PlotLegend -> {Sqrt[x], x, x^2, 2^x},
            LegendPosition -> {-0.7, 0.1} ];
```



8.4 Advanced Topic: A New Mathematical Function

Mathematica has many special functions built in. In this section, we look at what it takes to add another function to this collection.

8.4.1 Properties of Built-in Functions

For a built-in function, the code usually provides the following:

- Special values that are known exactly in terms of other functions.
- Expansion into power series.
- Numerical evaluation to any accuracy.
- Derivatives and indefinite integrals.

Let us look at the Bessel functions $J_n(x)$ or `BesselJ[n, x]`, for example.

For $n = 1/2$ a formula for the exact value is given.

```
In[1]:= BesselJ[1/2, x]

                2
        Sqrt[--] Sin[x]
               Pi
Out[1]= ---------------
            Sqrt[x]
```

This is a series representation of $J_1(x)$ at $x = 0$.

```
In[2]:= Series[BesselJ[1, x], {x, 0, 7}]

             3     5       7
        x   x     x       x            8
Out[2]= - - -- + --- - ----- + O[x]
        2   16   384   18432
```

20 digits of $J_2(17)$.

```
In[3]:= N[BesselJ[2, 17], 20]

Out[3]= 0.15836384123850347142
```

There is a formula for its derivative.

```
In[4]:= D[BesselJ[1, x], x]

        BesselJ[0, x] - BesselJ[2, x]
Out[4]= -----------------------------
                      2
```

There are formulae for the integrals of $J_n(x)$ for odd n.

```
In[5]:= Integrate[BesselJ[1, x], x]

                                2
Out[5]= -BesselJ[0, Sqrt[x ]]
```

Built-in functions come also with appropriate formatting rules for traditional form output. We shall discuss such formatting rules in Section 9.5.

8.4.2 Definitions for a New Function

We can provide much of the same functionality for other functions that are not built in. We need to know their mathematical properties, series representations, and special values.

Formulae are obtained from handbooks of mathematical functions, for example, Gradshteyn and Ryzhik or Abramowitz and Stegun. For an example of a function that is not built-in, let us turn to the *Struve functions* $H_\nu(z)$. These functions are closely related to the Bessel functions.

■ 8.4.2.1 Special Values

First, the special values. The handbooks contain the following formulae:

$$H_{n+\frac{1}{2}}(z) = Y_{n+\frac{1}{2}}(z) + \frac{1}{\pi}\sum_{m=0}^{n} \frac{\Gamma(m+\frac{1}{2})(\frac{z}{2})^{-2m+n-1/2}}{\Gamma(n+1-m)} \tag{8.4–1}$$

$$H_{-(n+\frac{1}{2})}(z) = (-1)^n J_{n+\frac{1}{2}}(z)\,. \tag{8.4–2}$$

These are easily turned into the following two definitions:

```
StruveH[r_Rational?Positive, z_] /; Denominator[r] == 2 :=
    BesselY[r, z] +
    Sum[Gamma[m + 1/2] (z/2)^(-2m + r - 1)/Gamma[r + 1/2 - m], {m, 0, r-1/2}]/Pi

StruveH[r_Rational?Negative, z_] /; Denominator[r] == 2 :=
    (-1)^(-r-1/2) BesselJ[-r, z]
```

Special values of Struve functions

Note that we match the index $n + 1/2$ in the form `r_Rational` with the condition `Denominator[r] == 2`. On the right side, n is expressed as $r - 1/2$ or $-r - 1/2$ in the second formula.

■ 8.4.2.2 Series Expansion

For the series expansion we find the following defining formula for $H_\nu(z)$:

$$H_\nu(z) = \sum_{m=0}^{\infty} (-1)^m \frac{(\frac{z}{2})^{2m+\nu+1}}{\Gamma(m+\frac{3}{2})\Gamma(\nu+m+\frac{3}{2})}\,. \tag{8.4–3}$$

For a power series of order n, we need to include terms up to $m = (n - \nu - 1)/2$. An expression in `z` is most easily turned into a series by adding a term `O[z]^(`n`+1)` to it. *Mathematica* then converts the whole expression into a series. We can pull the factor $(z/2)^{\nu+1}$ out of the summation. Here is the definition:

```
StruveH/: Series[StruveH[nu_?NumberQ, z_], {z_, 0, ord_Integer}] :=
    (z/2)^(nu + 1) Sum[ (-1)^m (z/2)^(2m)/Gamma[m + 3/2]/Gamma[m + nu + 3/2],
                       {m, 0, (ord-nu-1)/2} ] + O[z]^(ord+1)
```

Power series definition

If needed, other formulae can be developed for series at points other than 0. It might also be useful to have rules for dealing with series given as *arguments* of `StruveH[]`.

8.4.2.3 Numerical Evaluation

For numerical evaluation also, we can use Equation 8.4–3. We keep adding more terms until the result no longer changes. For this kind of power series with Γ-functions in the denominator this will be accurate enough.

```
SetAttributes[ StruveH, {NumericFunction, Listable} ]

StruveH[_, 0] := 0

StruveH[nu_?NumericQ, z_?NumericQ] /; Precision[{nu, z}] < Infinity :=
    Module[{s=0, so=-1, m=0},
        While[so != s,
            so = s;
            s += (z/2)^(2m+nu+1)/Gamma[m + 3/2]/Gamma[m + nu + 3/2];
            m++
        ];
        s
    ]
```

Numerical evaluation

In general, a good numerical definition should include different methods for different ranges of the values of the argument z. Often, it is also possible to estimate the number of terms needed beforehand and then use `Sum[]` instead of the `While[]` loop above. For functions with higher values of the index ν, there are often recurrence relations that express their values in terms of functions with lower index. The condition `Precision[{nu, z}] < Infinity` ensures that the rule is used only if at least one of the arguments ν and z is an approximate number. We do not want to give inexact results for exact inputs. We give a separate rule for the special case $z = 0$. We see from the definition of H_ν in the preceding section that the value of $H_\nu(0)$ is 0 for all values of ν.

The attribute `NumericFunction` enables a number of advanced features for the treatment of exact numeric quantities that are given as the arguments of `StruveH`, see Section 7.3.1.

8.4.2.4 Derivatives and Integrals

For computing derivatives we find the following formula:

$$H_{\nu-1}(z) - H_{\nu+1}(z) = 2H'_\nu(z) - \frac{(\frac{z}{2})^\nu}{\sqrt{\pi}\,\Gamma(\nu + \frac{3}{2})}\,. \tag{8.4–4}$$

This formula is easily programmed in *Mathematica*:

```
StruveH/: Derivative[0, n_Integer?Positive][StruveH] :=
  Function[{nu, z},
    D[ (StruveH[nu-1, z] - StruveH[nu+1, z] + (z/2)^nu/Sqrt[Pi]/Gamma[nu + 3/2])/2,
       {z, n-1} ]
  ]
```

Derivatives of Struve functions

No formulae exist to express indefinite integrals in terms of other known functions.

■ 8.4.2.5 A Performance Improvement

Summation formulae can often be speeded up by using incremental updates of the quantities involved. In our formula for the numerical values,

$$H_\nu(z) = \sum_{m=0}^{\infty} (-1)^m \frac{(\frac{z}{2})^{2m+\nu+1}}{\Gamma(m+\frac{3}{2})\Gamma(\nu+m+\frac{3}{2})} \tag{8.4–5}$$

we notice that for each m the numerator $(z/2)^{2m+\nu+1}$ can be computed from the previous one by multiplying it by $-(z/2)^2$ because the exponent increases by two for every term in the sum. The minus sign takes care of the change of signs of alternating terms expressed by the formula $(-1)^m$. The Γ-functions in the denominator have the nice property that $x\Gamma(x) = \Gamma(x+1)$ and so each successive term can be computed from the previous one without computing a single Γ function! We keep three variables `zf`, `g1`, and `g2` that hold the terms in the numerator and the two Γ-functions in the denominator and that are updated for each iteration.

```
StruveH[nu_?NumericQ, z_?NumericQ] /; Precision[{nu, z}] < Infinity :=
    Module[{s = 0, so = -1, z2 = -(z/2)^2, k1 = 3/2, k2 = nu + 3/2, g1, g2, zf},
        zf = (z/2)^(nu+1); g1 = Gamma[k1]; g2 = Gamma[k2];
        While[so != s,
            so = s; s += zf/g1/g2;
            g1 *= k1; g2 *= k2; zf *= z2; k1++; k2++
        ]; s
    ]
```

Faster numerical evaluation

■ 8.4.3 Putting Things Together

After having collected all the formulae we need, we put them into a complete package. Additionally, we make the function listable like any built-in one. The complete package Struve.m is reproduced in Listing 8.4–1. It contains additional definitions for typesetting Struve functions. These will be explained in Section 9.5.

Known special values are expressed in terms of Γ-functions for which in turn there are special values defined.

```
In[1]:= StruveH[ 1/2, x ]

              2            2
         Sqrt[--]     Sqrt[--] Cos[x]
              Pi           Pi
Out[1]= ---------  -  ---------------
         Sqrt[x]          Sqrt[x]
```

Here is an example of a power series for $H_2(x)$.

```
In[2]:= Series[ StruveH[2, x], {x, 0, 8} ]

            3        5          7
         2 x      2 x        2 x          9
Out[2]= ----- - ------ + -------- + O[x]
        15 Pi   315 Pi   14175 Pi
```

The first derivative is expressed again in terms of Struve functions according to our definition.

```
In[3]:= D[ StruveH[1, x], x ]

         2 x
         ---- + StruveH[0, x] - StruveH[2, x]
         3 Pi
Out[3]= -------------------------------------
                          2
```

A numerical value for $H_1(1.25)$.

```
In[4]:= StruveH[ 1, 1.25 ]

Out[4]= 0.298538
```

Here is a high-precision numerical value.

```
In[5]:= N[ StruveH[3, 1], 20 ]

Out[5]= 0.0058425265350728689O7
```

Because the argument and index of H_2 are exact numeric quantities, no numerical approximation takes place here.

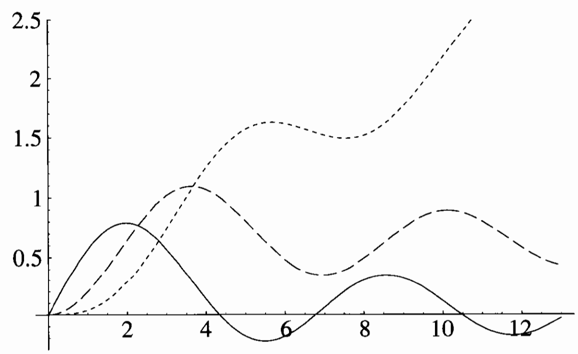
```
In[6]:= StruveH[ 2, Sqrt[2] ]

Out[6]= StruveH[2, Sqrt[2]]
```

But the attribute `NumericFunction` ensures that numerical approximation happens as soon as the expression comes into contact with an inexact number.

```
In[7]:= 1.0 * %

Out[7]= 0.109105
```

The numerical definitions given allow us to evaluate the Struve functions anywhere and, therefore, also plot them. A similar plot appears in Abramowitz and Stegun [2].

```
In[8]:= Plot[ {StruveH[0, x], StruveH[1, x], StruveH[2, x]},
              {x, 0, 13}, PlotRange -> {-0.3, 2.5},
              PlotStyle -> {Dashing[{}],
                            Dashing[{0.025, 0.01}],
                            Dashing[{0.008, 0.008}]} ];
```

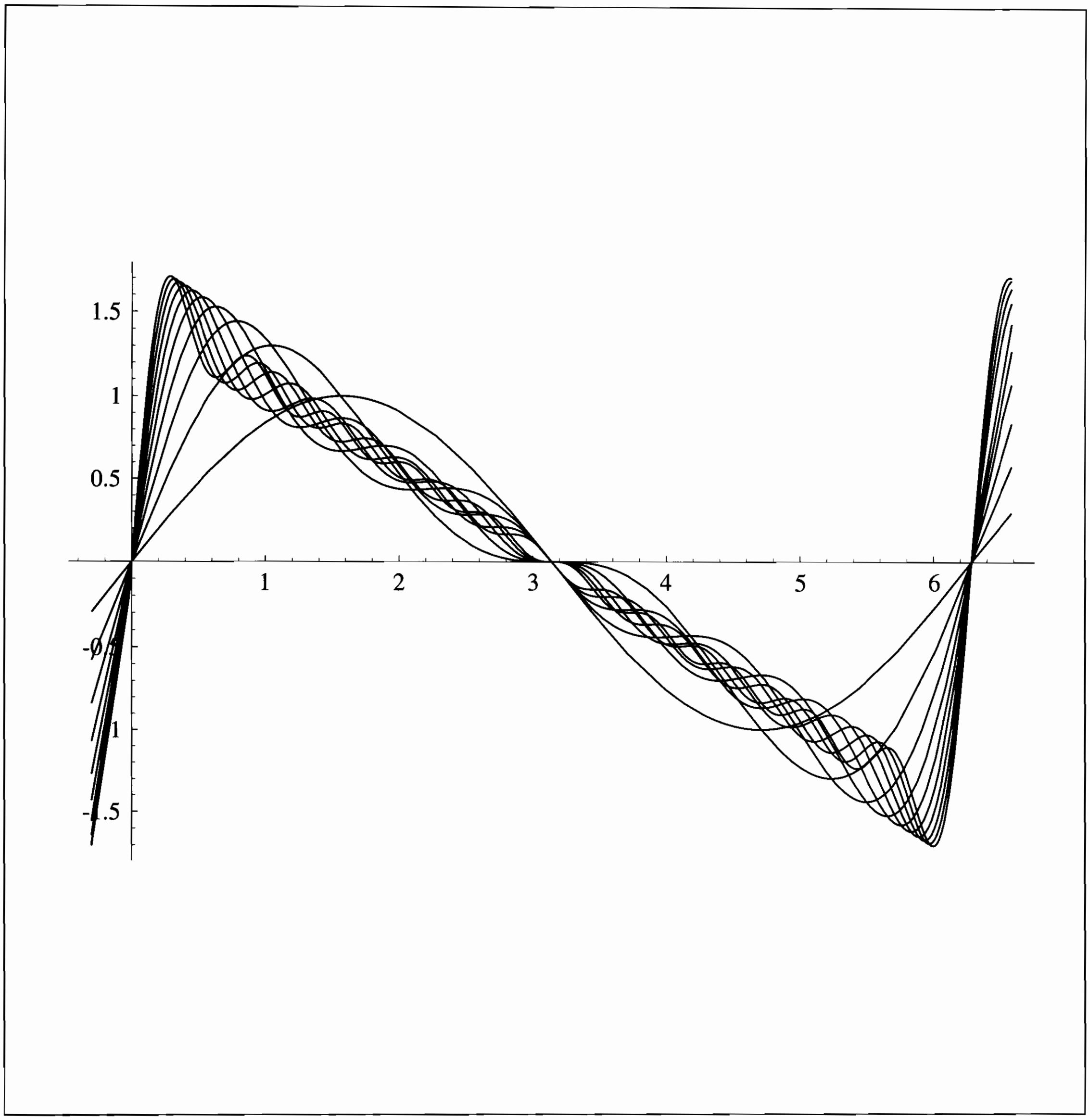


```
BeginPackage["ProgrammingInMathematica`Struve`"]

StruveH::usage = "StruveH[nu, z] gives the Struve function."

Begin["`Private`"]

SetAttributes[ StruveH, {NumericFunction, Listable} ]

(* special values *)

StruveH[r_Rational?Positive, z_] /; Denominator[r] == 2 :=
    BesselY[r, z] +
    Sum[Gamma[m + 1/2] (z/2)^(-2m + r - 1)/Gamma[r + 1/2 - m], {m, 0, r-1/2}]/Pi

StruveH[r_Rational?Negative, z_] /; Denominator[r] == 2 :=
    (-1)^(-r-1/2) BesselJ[-r, z]

(* Series expansion *)

StruveH/: Series[StruveH[nu_?NumberQ, z_], {z_, 0, ord_Integer}] :=
    (z/2)^(nu + 1) Sum[ (-1)^m (z/2)^(2m)/Gamma[m + 3/2]/Gamma[m + nu + 3/2],
                       {m, 0, (ord-nu-1)/2} ] + O[z]^(ord+1)

(* numerical evaluation *)

StruveH[_, 0] := 0

StruveH[nu_?NumericQ, z_?NumericQ] /; Precision[{nu, z}] < Infinity :=
    Module[{s = 0, so = -1, z2 = -(z/2)^2, k1 = 3/2, k2 = nu + 3/2, g1, g2, zf},
        zf = (z/2)^(nu+1); g1 = Gamma[k1]; g2 = Gamma[k2];
        While[so != s,
            so = s; s += zf/g1/g2;
            g1 *= k1; g2 *= k2; zf *= z2; k1++; k2++
        ]; s
    ]

(* derivatives *)

StruveH/: Derivative[0, n_Integer?Positive][StruveH] :=
  Function[{nu, z},
    D[ (StruveH[nu-1, z] - StruveH[nu+1, z] + (z/2)^nu/Sqrt[Pi]/Gamma[nu + 3/2])/2,
       {z, n-1} ]
  ]

(* interpretation and formatting for traditional form *)

StruveH/:
MakeBoxes[StruveH[nu_, z_], form:TraditionalForm] :=
    RowBox[{SubscriptBox["H", MakeBoxes[nu, form]], "(", MakeBoxes[z, form], ")"}]

MakeExpression[ RowBox[{SubscriptBox["H", nu_], "(", z_, ")"}],
                form:TraditionalForm ] :=
  MakeExpression[ RowBox[{"StruveH", "[", RowBox[{nu, ",", z}], "]"}], form ]

End[]

Protect[StruveH]

EndPackage[]
```

Listing 8.4–1: Struve.m: Definitions for the Struve functions

Chapter 9

Input and Output



In most programming languages, input and output are among the most tedious and difficult problems. The external form of a value often does not have much in common with the internal representation. As in graphics, there are many details that can be specified. *Mathematica*, as an interactive language, has certain input and output capabilities built in. It reads your input and displays it in a two-dimensional form on the screen, taking care of all the formatting. You can also save definitions and results to files without having to be concerned about formatting.

Only if you need to read files that are not written in *Mathematica*'s syntax or want to write files in other than *Mathematica*'s output form do you need to tell the system how to format your output. We can only treat a small number of the possibilities in this book. This, then, is a chapter on selected topics on input and output.

Section 1 is about formatting of output. It tells you how you can change the way expressions are formatted. Typesetting is not an easy thing to do, be it with TEX or with *Mathematica*. You will probably have to go through some trial and error to get it right.

In Section 2 we look at input from files and programs. This allows you to use *Mathematica* to analyze data obtained from other sources, laboratory measurements, for example.

Large calculations are the topic of Section 3. You can set things up so that *Mathematica* runs unattended, saving its results to a file rather than typing them to the screen.

In Section 4, we look at some ways of saving all your input and output in a file as you perform interactive calculations. Because notebooks are *Mathematica* expressions, we can rather easily write commands that save inputs and outputs in a notebook file.

The last section discusses the new typesetting features of Version 3. It explains the structure of typeset expressions and how you can define your own typeset forms for expressions.

About the illustration overleaf:

The first 10 partial sums of the Fourier series of the saw-tooth curve. Including more and more terms gets the approximations closer and closer to the limit curve.

```
l5 = Table[ Sum[Sin[i x]/i, {i, n}], {n, 10} ];
Plot[ Evaluate[l5], {x, -0.3, 2Pi+0.3} ]
```

9.1 Input and Output Formatting

The evaluation of expressions is done without regard to how expressions will eventually appear as output. After the evaluation is complete, the result is formatted for each output file to which it is to be printed. The *format type* of an output file specifies how expressions are to be printed. The most familiar of these format types is `OutputForm`, giving the usual two-dimensional rendering of output.

For each of the format types defined you can give rules that change the way certain expressions are rendered. The format types `InputForm`, `OutputForm`, `FullForm`, and so on are *text based*. The result of formatting is a string that is then output.

The two new format types for mathematical typesetting in Version 3, `StandardForm` and `TraditionalForm`, however, are based on *boxes*. The result of formatting is an expression consisting of a nested collection of box expressions. The frontend can interpret these boxes and display the expression in a two-dimensional active format. We shall discuss mathematical typesetting in Section 9.5. Here, we restrict ourselves to text-based input and output, relevant especially if you work with the kernel directly, or—more importantly—for output to files.

9.1.1 Format Types

A format type specifies how expressions are printed. You should be familiar with `OutputForm`, `InputForm`, and `FullForm`. By changing the format type of an output device or file, you can change the way in which expressions are written to that device. When you open a file for writing you can specify the format type with the option `FormatType`. The format type of an already open file, the standard output, for example, can be changed with `SetOptions[]`.

This gives the current values of all options of the standard output. The format type is `OutputForm`.

```
In[1]:= Options["stdout"]

Out[1]= {DOSTextFormat -> True, FormatType -> OutputForm,

  PageWidth -> 58, PageHeight -> 22,

  TotalWidth -> Infinity, TotalHeight -> Infinity,

  CharacterEncoding :> $CharacterEncoding,

  NumberMarks :> $NumberMarks}
```

Effective immediately, output appears in input form.

```
In[2]:= SetOptions["stdout", FormatType -> InputForm]

Out[2]= {DOSTextFormat -> True, FormatType -> InputForm,
   PageWidth -> 58, PageHeight -> 22,
   TotalWidth -> Infinity, TotalHeight -> Infinity,
   CharacterEncoding :> $CharacterEncoding,
   NumberMarks :> $NumberMarks}

In[3]:= Expand[(a + b)^2]

Out[3]= a^2 + 2*a*b + b^2
```

This is the input form of the output form.

```
In[4]:= OutputForm[%]

                       2            2
Out[4]"//"OutputForm= a  + 2 a b + b
```

Used as a command, a format type formats its argument in the required way, for example, `FullForm[`*expr*`]`. The result of this formatting is then rendered according to the format type of the output file, as we have just seen in the last example above. The list of available format types is assigned to the global variable `$PrintForms`. It is a *read-only* variable.

This is the list of format types available in the current version of *Mathematica*.

```
In[1]:= $PrintForms
Out[1]= {InputForm, OutputForm, TextForm, CForm,
  FortranForm, TeXForm, StandardForm, TraditionalForm}
```

■ 9.1.2 Defining Print Forms

For each format type, you can make definitions to change the way a particular expression is printed on an output file that uses that format type by giving rules of the form

`Format[`*pattern*, *formattype*`] :=` *expression*,

where *formattype* is a valid format type, and *expression* is an arbitrary *Mathematica* expression whose value is used to format any expression that matches *pattern*. *expression* will typically contain some formatting commands (see below). If *formattype* is missing, `OutputForm` is assumed. Such rules for formatting are stored with the head of *pattern* and not with the symbol `Format`.

For example, many mathematical functions have arguments that are normally printed as indices. The function `BesselJ[`n`, `x`]` is typeset as $J_n(x)$ in traditional mathematics. To have *Mathematica* print an approximation of this, use the definition

`Format[b:BesselJ[_, _]] := Subscripted[b, 1].`

System symbols for which formats are to be defined must be unprotected first.

```
In[1]:= Unprotect[BesselJ, BesselY];
```

Because we do not need the names of the arguments, we only give a name to the whole pattern.

```
In[2]:= Format[b:BesselJ[_, _]] := Subscripted[b, 1]
```

The format is used to print the first argument as a subscript.

```
In[3]:= BesselJ[1, x]
Out[3]= BesselJ [x]
               1
```

You can change the head of the printed expression to anything.

```
In[4]:= Format[BesselY[n_, x_]] := Subscripted[Y[n, x], 1]
```

`BesselY` now prints as `Y`, and with a subscript.

```
In[5]:= BesselY[2, z]
Out[5]= Y [z]
         2
```

The building blocks you can use to define formats are described in Subsections 2.8.6 and 2.8.8 of the *Mathematica* book. The formatting command `Subscripted[]` can take care of all the simpler cases, in which certain arguments are to be printed as subscripts or superscripts. In more complicated circumstances, you must assemble the output form using the primitive operations available.

■ 9.1.3 Application: Tensors

Tensors are a generalization of vectors and matrices used in physics, mechanical engineering, and mathematics. Here we are only concerned with typesetting problems for tensors. In the usual notation one might encounter things like ${\Gamma_k}^{ij}(x, y, z, t)$ or ${R_{ij}}^{kl}$. First, we have to design an input syntax, a way of typing such tensors in *Mathematica*. A possible way is

```
Tensor[Gamma][li[k], ui[i], ui[j]][x, y, z, t]
```

for ${\Gamma_k}^{ij}(x, y, z, t)$ and

```
Tensor[R][li[i], li[j], ui[k], ui[l]]
```

for ${R_{ij}}^{kl}$. We use the tags `ui[]` and `li[]` to denote upper and lower indices. Some way of indicating which expressions are tensors is necessary. We use `Tensor[`*h*`]` to denote the tensor h. To have these tensors print out in the desired way, we have to do some programming. (This is one of the more complicated circumstances mentioned in the previous subsection.) The result is shown in Listing 9.1–1.

```
BeginPackage["ProgrammingInMathematica`Tensors`"]

ui::usage = "ui[index] denotes an upper index in a tensor."
li::usage = "li[index] denotes a lower index in a tensor."
Tensor::usage = "Tensor[h][indices] denotes a tensor h with index list indices."

Begin["`Private`"]

Format[ Tensor[t_][ind___] ] :=
    Module[{indices},
        indices = {ind} /. {ui->Superscript, li->Subscript};
        SequenceForm[t, Sequence @@ indices]
    ]

End[]

Protect[ Tensor, ui, li ]

EndPackage[]
```

Listing 9.1–1: Tensors.m: Formatting tensors

The substitution builds a list of subscripts and superscripts. This list is then spliced into the argument list of `SequenceForm[]` which prints its arguments without intervening space.

Here is our first test example.

```
In[1]:= Tensor[Gamma][li[k], ui[i], ui[j]][x, y, z, t]

                 ij
Out[1]= Gamma      [x, y, z, t]
                k
```

And here is the second one.

```
In[2]:= Tensor[R][li[i], li[j], ui[k], ui[l]]

             kl
Out[2]= R
           ij
```

Rules belong to a certain format type. We have not defined a rule for the format type `TeXForm` and therefore no special formatting is done (see Exercise 8).

```
In[3]:= TeXForm[ Tensor[g][ui[i], li[j]] ]

Out[3]//TeXForm=
  \Muserfunction{Tensor}(g)(\Muserfunction{ui}(i),
   \Muserfunction{li}(j))
```

Note that we have defined a rule only for objects of the form `Tensor[`*t*`][`*indices*`]`. The first of our examples is of the form `Tensor[`*t*`][`*indices*`][`*arguments*`]`. The *head* of this expression matches the rule for formatting and the elements are then formatted in the default way.

■ 9.2 Input from Files and Programs

■ 9.2.1 Low-Level Input

The terms "low-level" and "high-level" refer to the amount of details you have to program yourself. In traditional programming languages, files are usually treated at a rather low level. You open a file, perform various read operations, and then close it again. In *Mathematica*, these functions are `OpenRead["`*file*`"]`, `Read["`*file*`"]`, and `Close["`*file*`"]`.

Open files (or other external objects, such as pipes) are called *streams*. As *Mathematica* objects, they are represented as `InputStream[]`, `OutputStream[]`, or `LinkStream[]`. Opening a file (by `OpenRead[]`, `OpenWrite[]`, or `OpenAppend[]`) returns such a stream object. This object should be used for all further references to the open file. A typical program segment that reads *Mathematica* expressions from a file is shown in Listing 9.2–1.

```
ReadLoop[fileName_String] :=
    Module[{file, expr},
        file = OpenRead[fileName];
        If[ file === $Failed, Return[file] ];
        While[ True,
            expr = Read[file];
            If[ expr === EndOfFile, Break[] ];
            Print["expr is ", expr]
        ];
        Close[file]
    ]
```

Listing 9.2–1: ReadLoop1.m: A simple loop for reading expressions

The command `ReadLoop[]` receives the name of the file to read as parameter. It then opens that file. `OpenRead[]` returns a stream, which we assign to the local variable `file`. The following `While[]` loop reads expressions from the open file and prints them out until it encounters the end of the file, indicated by the symbol `EndOfFile`. Finally, we close the file.

With external operations there is always a chance for failure, the most common being that the file simply does not exist. In such a case, `OpenRead[]` prints a message and returns the symbol `$Failed` instead of the stream. A good program should test for this return value. In case of an error, the function returns prematurely. There is no need to print an error message; this has already been done by `OpenRead[]`.

■ 9.2.2 Referring to Open Files

The reason for using the return value of `OpenRead[]` for all future references to the file, rather than its original name `fileName`, is that it is not always possible to uniquely identify an open file from its name. Most operating systems provide abbreviated ways of referring

to files, and *Mathematica* takes these into account when opening a file. The file name inside the stream returned by `OpenRead[]` is the fully expanded "absolute" file name. It is under this name that *Mathematica* stores the information associated with all open files it maintains (`Streams[]` returns the list of all open files).

The syntax ~ refers to a user's home directory in UNIX. *Mathematica* expands this name to the absolute path name shown.

```
In[1]:= file = OpenRead[ "~/init.m" ]

Out[1]= InputStream[/home/bellatrix/maeder/init.m, 4]
```

It does not find the file under its original name and gets rather confused.

```
In[2]:= Read[ "~/init.m" ]

General::aofil:
   ~/init.m already open as /home/bellatrix/maeder/init.m.

Read::openx: ~/init.m is not open.

Out[2]= Read[~/init.m]
```

Always use the name returned by `OpenRead[]`. (The init.m file contains commands; when read, they usually return `Null`, which is not printed.)

```
In[3]:= Read[ file ]
```

At the end, the file should be closed. `Close[]` returns the full name of the file closed.

```
In[4]:= Close[ file ]

Out[4]= /home/bellatrix/maeder/init.m
```

■ 9.2.3 Things to Read

`Read[`*file*`]` reads an expression. `Read[`*file*`, `*type*`]` with a second argument can be used to read other things as well.

`Byte`	a single byte
`Character`	a character, returned as a string
`Word`	a string, delimited by white space
`String`	a line, returned as a string
`Real`	floating-point number in FORTRAN form
`Number`	number in FORTRAN form
`Expression`	a *Mathematica* expression

Types of things to read; see also Subsection 2.11.7 of the *Mathematica* book

The most interesting feature is the ability to give a *skeleton expression* as the thing to read. All occurrences of the basic types `Expression`, `Number`, ... in this skeleton will be filled from the file and the resulting expression is then returned. The file datafile used in the next example contains some numbers on separate lines to demonstrate these ideas.

```
55
66
1.11111111
1
2
3
777
```

datafile: Sample input file

First, we open the file for reading.

```
In[1]:= file = OpenRead["datafile"]
Out[1]= InputStream[datafile, 4]
```

This reads an integer as a *Mathematica* integer.

```
In[2]:= Read[ file, Number ]
Out[2]= 55
```

`Real` converts numbers to approximate numbers.

```
In[3]:= Read[ file, Real ]
Out[3]= 66.
```

The number read is inserted in place of the symbol `Number` in the skeleton given.

```
In[4]:= Read[ file, f[Number] ]
Out[4]= f[1.11111111]
```

Here are three slots to fill and so three numbers are read.

```
In[5]:= Read[ file, {Number, Number, Number} ]
Out[5]= {1, 2, 3}
```

■ 9.2.4 Application: Reading Unevaluated Expressions

When you read an expression, either by `Read[`*file*`]` or `Read[`*file*`, Expression]`, it is evaluated as part of the normal evaluation sequence. To return an unevaluated expression, you need to somehow wrap `Hold[]` or `HoldForm[]` around it before it gets a chance of being evaluated. The observations made in the previous subsection show a way of doing this. Simply use `Read[`*file*`, Hold[Expression]]`. *Mathematica* will fill in an expression from the file in place of the keyword `Expression` and because it is inside `Hold[]` it will not be evaluated later on. The read loop ReadLoop2.m, shown in Listing 9.2–2 reads expressions from a file and prints them out—unevaluated.

```
ReadLoop[fileName_String] :=
    Module[{file, expr},
        file = OpenRead[fileName];
        If[ file === $Failed, Return[file] ];
        While[ True,
            expr = Read[file, HoldForm[Expression]];
            If[ expr === EndOfFile, Break[] ];
            Print[ ]; Print["expr is ", expr]
        ];
        Close[file]
    ]
```

Listing 9.2–2: ReadLoop2.m: Reading expressions unevaluated

Here is this function applied to the package ExpandBoth.m from Section 2.1.3. The individual commands in that file are read as expressions, but they are not evaluated. `HoldForm[]` is like `Hold[]`, but it is invisible on output.

The value returned by `ReadLoop[]` is the value of the `Close[]` command, which returns the name of the file closed.

```
In[1]:= ReadLoop["ExpandBoth.m"]
expr is ExpandBoth::usage =
  ExpandBoth[e] expands all numerators and denominators\
   in e.
expr is Begin[`Private`]
expr is ExpandBoth[x_Plus] := ExpandBoth /@ x
                              Expand[Numerator[x]]
expr is ExpandBoth[x_] := ----------------------
                             Expand[Denominator[x]]
expr is End[]
expr is Null
Out[1]= ExpandBoth.m
```

■ 9.2.5 Reading from a Program

Under any reasonable operating system it is possible to read from an external program in the same way that it is possible to read from a file. Opening the file starts the external program and every read statement will wait for the program to supply enough output to satisfy the request. Closing the file terminates the external program.

Here is a small program that runs the UNIX date command to obtain the current date and time (Listing 9.2–3).

```
UnixDate[] :=
    Module[{process, result},
        process = OpenRead["!date"];
        result = Read[process, String];
        If [ result === EndOfFile, Return[$Failed] ];
        Close[process];
        result
    ]
```

Listing 9.2–3: UnixDate1.m: the first version of the `UnixDate[]` command

The output of the date command is returned as a string.

```
In[1]:= UnixDate[]
Out[1]= Sat Oct  5 20:44:21 MET DST 1996
```

Error checking is different in this case. Opening an external program always succeeds, even if the program does not exist. If the program does not exist, then the first attempt to read from it will return `EndOfFile` and that is the condition we test.

As a string, the date is not very useful. The output format of the date command is under program control (through command line options, not available in all versions of UNIX) and we can get it to print the date in a form that is acceptable to *Mathematica*, as a list of numbers! We then read it as an `Expression` instead of a `String`. The code is shown in Listing 9.2–4. (Note that it will fail starting January 1, 2000.)

```
UnixDate[] :=
    Module[{process, result},
        process = OpenRead["!date '+{19%y, %m, %d, %H, %M, %S}'"];
        result = Read[process, Expression];
        If [ result === EndOfFile, Return[$Failed] ];
        Close[process];
        result
    ]
```

Listing 9.2–4: UnixDate2.m: the second version of the `UnixDate[]` command

The output of the date command is returned as a list {*year*, *month*, *day*, *hour*, *minute*, *second*}.

```
In[1]:= UnixDate[]
Out[1]= {1996, 10, 5, 20, 44, 30}
```

The built-in command `Date[]` does essentially the same thing.

```
In[2]:= Date[]
Out[2]= {1996, 10, 5, 20, 44, 31}
```

■ 9.2.6 High-Level Input

High-level input takes care of the details of opening and closing files by itself. The most commonly used input function is undoubtedly `Get["`*file*`"]`, usually used in its prefix form `<<`*file*.

Another useful command is `ReadList[]`, which is the way to transfer data from other programs or laboratory measurements into *Mathematica*. It reads the whole file and returns a list of all the things read. The things to read are specified in the same way as in `Read[]` (Section 9.2.1). In fact, it is an easy exercise to write `ReadList[]` in terms of `Read[]`; see Listing 9.2–5.

```
MyReadList[fileName_String, thing_:Expression] :=
    Module[{file, expr, list = {}},
        file = OpenRead[fileName];
        If[ file === $Failed, Return[file] ];
        While[ True,
            expr = Read[file, thing];
            If[ expr === EndOfFile, Break[] ];
            AppendTo[list, expr]
        ];
        Close[file];
        list
    ]
```

Listing 9.2–5: ReadList.m: Our own `ReadList[]` command

It works just like `ReadList[]`, reading all the expressions in the file (they are all numbers) and collecting them in a list.

```
In[1]:= MyReadList[ "datafile" ]
Out[1]= {55, 66, 1.11111111, 1, 2, 3, 777}
```

The contents of the file are read as approximate numbers that are inserted as arguments of `Sin[]`, which then evaluates to the sine of the numbers in the file.

```
In[2]:= MyReadList[ "datafile", Sin[Real] ]
Out[2]= {-0.999755, -0.0265512, 0.896192, 0.841471,
   0.909297, 0.14112, -0.855551}
```

■ 9.2.7 High-Level Program Input

Again, the file to read can be a program. With the two commands `Get["!`*program*`"]` and `ReadList["!`*program*`"]` we do not have to start and terminate the program explicitly. Our `UnixDate[]` function from Section 9.2.5 now becomes a one-liner:

```
UnixDate[] := << "!date '+{19%y, %m, %d, %H, %M, %S}'".
```

9.3 Running *Mathematica* Unattended

Mathematica is quite able to perform calculations that run for hours or even days. If you want to perform a longer computation, then you might want to run it in the background. This section gives some hints for doing this under UNIX. It does not apply to PC-style machines where there is not much point in setting things up differently. You just let *Mathematica* run for as long as you wish.

9.3.1 Batch Mode

This subsection gives some ideas on how to do large calculations unattended, often called *batch mode*. Much of this material is specific to the operating system UNIX. It has been tested on a SPARCstation under Version 4.1 of SunOS and under Solaris 2.5.1 using the C-shell. It is trivial to adapt this to other versions of UNIX and it should also be possible to do the same kind of things on other multi-tasking operating systems.

The main difference between the normal interactive mode of operation and batch mode is that you do not have a chance to react to things that go wrong. All your commands have to be set up in advance in a file from which *Mathematica* will then read its commands. Having solved a smaller problem interactively, you can review your interactive session and assemble the commands in an input file for a later batch run. This file is then given to *Mathematica* as input instead of your keyboard as usual.

The output that normally appears on the screen can be captured in an output file. After the computation is complete, you can look at this file for any error conditions and hopefully find the answer to your problem in it.

The following computation will depend on a parameter n that we can later increase for the larger batch mode calculation.

```
In[1]:= n = 3
Out[1]= 3
```

We use the package mentioned in Section 5.5.4.2.

```
In[2]:= << SwinnertonDyer.m
```

We compute the n^{th} Swinnerton-Dyer polynomial.

```
In[3]:= SwinnertonDyerP[n, x]
                    2         4        6    8
Out[3]= 576 - 960 x  + 352 x  - 40 x  + x
```

We are interested in how long it takes to prove it irreducible by trying to factor it.

```
In[4]:= Timing[ Factor[%] ]
                                        2         4        6    8
Out[4]= {0.05 Second, 576 - 960 x  + 352 x  - 40 x  + x }
```

To prepare for a longer computation, with $n = 5$ say, we collect the commands we just entered into a file sw5.in:

```
<< SwinnertonDyer.m
n = 5
SwinnertonDyerP[n, x];
Timing[ Factor[%] ]
```

and then issue the following command at the shell prompt:

```
nice +16 math < sw5.in >& sw5.log &
```

This starts *Mathematica* in the background at a lower priority, connecting the standard input to the file sw5.in and capturing all output in sw5.log. After the computation has finished, the log file will look like this:

```
Mathematica 3.0 for SPARC
Copyright 1988-96 Wolfram Research, Inc.

In[1]:=
In[2]:=
Out[2]= 5

In[3]:=
In[4]:=
                                                            2
Out[4]= {20.95 Second, 2000989041197056 - 44660812492570624 x  +

                           4                     6                      8
>      183876928237731840 x  - 255690851718529024 x  + 172580952324702208 x  -

                          10                     12                      14
>      65892492886671360 x   + 15459151516270592 x   - 2349014746136576 x   +

                        16                  18                   20
>      239210760462336 x   - 16665641517056 x   + 801918722048 x   -

                    22              24             26          28        30
>      26625650688 x   + 602397952 x   - 9028096 x   + 84864 x   - 448 x   +

        32
>      x  }

In[5]:=
```

You notice that the input lines are empty. Normally the operating system echoes the input back to the terminal. Here you want *Mathematica* itself to echo all input to the standard output. The variable `$Echo` is a list of files to which input is echoed. It is normally empty. To have your input appear in the output file, put the following command at the beginning of your input file:

```
AppendTo[ $Echo, "stdout" ]; $Line--;.
```

This also resets the line numbers so that the rest of the calculations starts with input line 1 as before (this is optional, of course).

It might be useful to have *Mathematica* print its output in InputForm so it could be input into another computation easily. You can either set $PrePrint = InputForm (see Section 8.1.1), or you can change the format type of the standard output with

SetOptions["stdout", FormatType -> InputForm].

■ 9.3.2 Infinite Calculations: The Collatz Sequence

An infinite calculation is one that you can run for as long as you wish (or until the computer crashes) and that produces one result after another. You can use this to try to find counterexamples to a conjecture or to find numbers n with a certain property by testing for $n = 1, 2, 3, \ldots$ for as long as your patience lasts.

In *Mathematica* you write a short program that contains an infinite loop and repeatedly does some computation. Because it potentially never returns (you have to interrupt the computation to get back at the next input prompt), you should use Print[] statements inside the loop to inform yourself about the progress of the computation. For an example, let us look at the famous $3n + 1$ problem, also called the Collatz problem. Starting with an integer k_1, we construct a sequence of integers $k_1, k_2, k_3, \ldots$ according to the following formula:

$$k_{i+1} = \begin{cases} (3k_i + 1)/2, & k_i \text{ odd;} \\ k_i/2, & k_i \text{ even.} \end{cases} \qquad (9.3\text{–}1)$$

If you try this out, you will notice that after a while one of the k_i becomes 1 and then the sequence repeats itself with $1, 2, 1, 2, \ldots$.

Note that we changed the definition of the Collatz sequence from earlier editions of this book. The old formula used $k_{i+1} = 3k_i + 1$, for odd k_i. Defined in this way, k_{i+1} is always even and the next thing we do is divide it by two.

We want to find the integer with the longest sequence before reaching 1. The function StoppingTime[*n*] computes this length, called the *total stopping time:*

```
c[ k_?EvenQ ] := k/2
c[ k_ ] := (3k + 1)/2

StoppingTime[k_Integer?Positive] :=
    Module[{i=1, m=k}, While[m != 1, m = c[m]; i++]; i ]
```

The length of the Collatz sequence

Now we write our infinite loop. It computes StoppingTime[*i*] for i from some lower bound on upward. Whenever it finds a new maximum, it prints a line. The only way to stop it is to interrupt *Mathematica* or to kill the process.

```
FindMaxima[low_] :=
    Module[{m=0, k=0, i=low, si},
        While[ True,
            si = StoppingTime[i];
            If[si > m, {m, k} = {si, i}; Print["StoppingTime[", k, "] = ", m] ];
            i++;
        ]
    ]
```

Part of Collatz.m: Finding maximal stopping times

Starting with 27, it takes 71 steps before reaching 1.

```
In[1]:= StoppingTime[ 27 ]

Out[1]= 71
```

We start the computation at 1 and let it run for a while before interrupting it.

```
In[2]:= FindMaxima[ 1 ]

StoppingTime[1] = 1
StoppingTime[2] = 2
StoppingTime[3] = 6
StoppingTime[6] = 7
StoppingTime[7] = 12
StoppingTime[9] = 14
StoppingTime[18] = 15
StoppingTime[25] = 17
StoppingTime[27] = 71
StoppingTime[54] = 72
StoppingTime[73] = 74
StoppingTime[97] = 76
StoppingTime[129] = 78
StoppingTime[171] = 80
StoppingTime[231] = 82
StoppingTime[313] = 84
StoppingTime[327] = 92
StoppingTime[649] = 93
StoppingTime[703] = 109
StoppingTime[871] = 114
StoppingTime[1161] = 116
^C
```

This kind of computation is, of course, well suited for running in batch mode, as explained in Section 9.3.1. The following command file collatz.in could be used:

```
AppendTo[ $Echo, "stdout" ]; $Line--;
<< Collatz.m
FindMaxima[1]
```

Performing the computation in the background and redirecting its output to the file collatz.log, you could then examine the output file from time to time to see how far the computation has progressed.

Apart from printing the maxima as they are found, it is also a good idea to print some information about the progress of the computation. Should the computer crash while you are running your computation, you would not know where to restart it and would have to go back to the last maximum found if no progress report had been generated. The following variant of `FindMaxima[]` prints the value of the loop variable after every 100 iterations.

```
FindMaxima[low_] :=
    Module[{m=0, n=0, i=low, si},
        While[ True,
            si = StoppingTime[i];
            If[si > m, {m, n} = {si, i}; Print["StoppingTime[", n, "] = ", m] ];
            i++;
            If[ Mod[i, 100] == 0, Print["i = ", i] ]
        ]
    ]
```

Keeping informed about progress

Should the computation abort abnormally, you can simply restart it at the last value reported. This is the reason we provided for the argument `low` in the definition of `FindMaxima[]`.

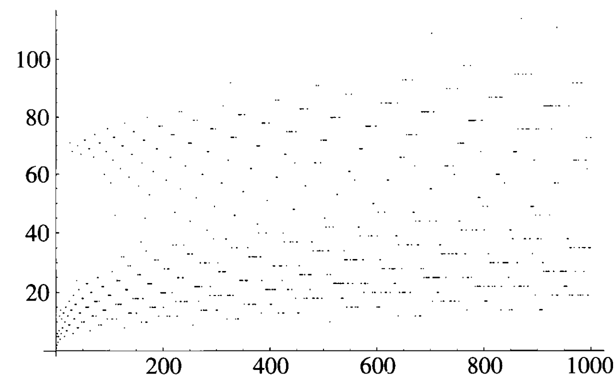
If you really want to hunt for new records then it pays to look more closely at the definition of the Collatz sequence. It does not make much sense, for example, to check the length for an even integer because the first thing we do is divide it by two. We could therefore always start with an odd number and increase the loop variable `i` by 2 in each iteration. The `StoppingTime[]` function is indeed rich in interesting patterns. Some of these can be revealed by a simple plot of its values for the first few hundred integers.

We compute the first 1000 values of the stopping time.

```
In[3]:= vals = Array[StoppingTime, 1000];
```

And then we draw a dot for each of its values.

```
In[4]:= ListPlot[ vals, PlotRange -> All ];
```



The standard package Examples/Collatz.m contains more sophisticated methods to speed up the computation of the stopping time.

■ 9.3.3 Application: Efficient Computation of the Collatz Sequence

An interesting exercise in efficient programming is the construction of the Collatz sequence itself. Equation 9.3–1 is easy to code:

```
c[ k_?EvenQ ] := k/2
c[ k_ ] := (3k + 1)/2
```

The generator of the Collatz sequence

If we knew the length of the sequence (until it reaches 1 for the first time), we could use `NestList[c, `k_1`, `n`]`, but it is not even known whether the sequence terminates in 1 for every input k_1. The first attempt uses `AppendTo[]` to construct the sequence iteratively:

```
Collatz1[k1_Integer?Positive] :=
    Module[{seq = {k1}, k = k1},
        While[ k != 1, k = c[k]; AppendTo[seq, k] ];
        seq
    ]
```

The stopping times of these numbers are 10, 20, 30, ... 400. We can use these data to measure the performance of the various methods for computing the Collatz sequence.

```
In[1]:= tens =
      {17, 65, 270, 379, 593, 91, 41, 171, 731, 1017,
       1406, 5921, 8315, 11676, 5567, 7963, 10971, 45055,
       64255, 89119, 123391, 173321, 254911, 351359,
       493537, 701607, 980905, 461262, 1909305, 910107,
       1302127, 626331, 837799, 3558763, 1723519, 6956969,
       9447698, 13270838, 6674175, 27988635};
```

This auxiliary function measures the timings of *coll*[*k*] for method *coll* applied to a list of *k* values.

```
In[2]:= timings[ coll_, ks_ ] :=
        Function[k, Timing[coll[k]][[1]] /. Second->1] /@
        ks
```

Here we measure the timings for `Collatz1[]`. They are quadratic in the length of the sequence.

```
In[3]:= (t1 = timings[ Collatz1, tens ]) // Short
Out[3]//Short=
 {0.02, 0.03, 0.05, 0.05, 0.08, 0.09, 0.11, 0.12, 0.14,
  0.15, <<25>>, 0.7, 0.73, 0.74, 0.77, 0.79}
```

An alternative is a recursive computation, using `Prepend[]`:

```
Collatz2[1] = {1}
Collatz2[k_Integer?Positive] := Prepend[ Collatz2[c[k]], k ]
```

This code is rather elegant, but not faster than the first method. Because it is recursive, it uses up stack space.

The deep recursion requires setting `$RecursionLimit` to infinity.

```
In[4]:= $RecursionLimit = Infinity
Out[4]= Infinity
```

Here we measure the timings for `Collatz2[]`. The timings are almost identical to those of `Collatz1[]`.

```
In[5]:= (t2 = timings[ Collatz2, tens ]) // Short
Out[5]//Short=
 {0.01, 0.03, 0.04, 0.05, 0.07, 0.08, 0.11, 0.11, 0.12,
  <<26>>, 0.67, 0.69, 0.71, 0.74, 0.75}
```

There is one way to avoid the slow append or prepend operations: building up a deeply nested list and then flattening it in one step:

```
Collatz[k_Integer?Positive] := Flatten[ appendCollatz[{}, k] ]

appendCollatz[sofar_, 1] := {sofar, 1}
appendCollatz[sofar_, k_Integer] := appendCollatz[{sofar, k}, c[k]]
```

Here we see the nested list built by the auxiliary function appendCollatz, starting with an empty list.

```
In[6]:= appendCollatz[{}, 17]
Out[6]= {{{{{{{{{{{}, 17}, 26}, 13}, 20}, 10}, 5}, 8},
    4}, 2}, 1}
```

One fast flattening operation puts it into the required form.

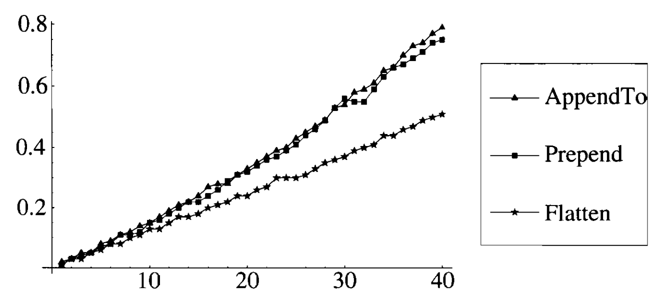
```
In[7]:= Flatten[ % ]
Out[7]= {17, 26, 13, 20, 10, 5, 8, 4, 2, 1}
```

Here we measure the timings for the fastest version, Collatz[]. Timings increase linearly with the length of the sequence.

```
In[8]:= (t3 = timings[ Collatz, tens ]) // Short
Out[8]//Short=
 {0.01, 0.03, 0.03, 0.05, 0.06, 0.08, 0.08, 0.1, 0.11,
  0.13, <<25>>, 0.46, 0.47, 0.49, 0.5, 0.51}
```

This graphic shows the three timings in comparison. The graphic command is from the standard package Graphics/MultipleListPlot.m.

```
In[9]:= MultipleListPlot[ t1, t2, t3, PlotJoined->True,
          SymbolShape->{PlotSymbol[Triangle, 1.5],
                        PlotSymbol[Box, 1],
                        PlotSymbol[Star, 1.6]},
          PlotLegend->{AppendTo, Prepend, Flatten} ];
```



Listing 9.3–1 shows the complete code of the package Collatz.m, containing all of the functions mentioned in this section.

```
BeginPackage["ProgrammingInMathematica`Collatz`"]

Collatz::usage = "Collatz[n] gives a list of the iterates in the 3n+1 problem,
    starting from the positive integer n, until reaching 1 for the first
    time. The conjecture is that this sequence always terminates."

StoppingTime::usage = "StoppingTime[n] finds the total stopping time of the
    integer n. This is the length of the Collatz sequence before hitting 1."

FindMaxima::usage = "FindMaxima[from] reports successive maxima of the
    total stopping time starting the search with the integer from."

Begin["`Private`"]

c[ k_?EvenQ ] := k/2
c[ k_ ] := (3k + 1)/2

Collatz[ k_Integer?Positive ] := Flatten[ appendCollatz[{}, k] ]

appendCollatz[sofar_, 1] := {sofar, 1}
appendCollatz[sofar_, k_Integer] := appendCollatz[{sofar, k}, c[k]]

StoppingTime[ k_Integer?Positive ] :=
    Module[{i=1, m=k}, While[m != 1, m = c[m]; i++]; i ]

FindMaxima[ low_ ] :=
    Module[{m=0, k=0, i=low, si},
        While[ True,
            si = StoppingTime[i];
            If[si > m, {m, k} = {si, i}; Print["StoppingTime[", k, "] = ", m] ];
            i++;
            If[ Mod[i, 100] == 0, Print["i = ", i] ]
        ]
    ]

End[]

Protect[ Collatz, StoppingTime, FindMaxima ]

EndPackage[]
```

Listing 9.3–1: Collatz.m: Computations with the Collatz sequence

■ 9.4 Session Logging

This section shows some ways of recording the input or output of a *Mathematica* session in a file. If you use *Mathematica* with the notebooks frontend, all input and output is saved in a notebook and you may not have to use the mechanisms explained in this section, except in special circumstances.

■ 9.4.1 Keeping a Log of the Input

The standard package Utilities/Record.m contains the single command

```
AppendTo[ $Echo, OpenAppend["math.record"] ],
```

which will cause all input to be written to the file math.record.

An output channel such as `$Echo` is a list of files or streams to which certain output is written. A list of all such channels is in Subsection 2.13.1 of the *Mathematica* book. By default, `$Echo` is the empty list and *Mathematica* does not echo its input, as we have seen in Section 9.3.1.

■ 9.4.2 Logging Input and Output

If you want to keep a record of *Mathematica*'s output, you can similarly use

```
AppendTo[ $Output, OpenWrite["math.log"] ].
```

The default format type for a newly opened file is `InputForm`; therefore, the record will be in a form that is easy to read back into *Mathematica* later on.

`$Output` is of course not empty to begin with. Output is now written to two files, one being the standard output, the other one the just-opened math.log.

```
In[1]:= AppendTo[ $Output, OpenWrite["math.log"] ]
Out[1]= {OutputStream[stdout, 1],
  OutputStream[math.log, 4]}
```

Here we generate some output.

```
In[2]:= Expand[ (a+b)^2 ]
             2            2
Out[2]= a  + 2 a b + b

In[3]:= Factor[ x^5-1 ]
                                 2    3    4
Out[3]= (-1 + x) (1 + x + x  + x  + x )
```

After this session, math.log contains the following text:

```
Out[1]= {OutputStream["stdout", 1], OutputStream["math.log", 4]}

In[2]:=
Out[2]= a^2 + 2*a*b + b^2

In[3]:=
Out[3]= (-1 + x)*(1 + x + x^2 + x^3 + x^4)

In[4]:=
```

We notice that it also contains the input and output prompts (these prompts are printed just like other output and therefore they appear in all files of the channel `$Output`). Perhaps we can put both, input and output, into the same file? Listing 9.4–1 shows a solution.

```
BeginPackage["ProgrammingInMathematica`SessionLog`"]

OpenLog::usage = "OpenLog[filename, opts..] starts logging all input and output
    to filename."
CloseLog::usage = "CloseLog[] closes the logfile opened by OpenLog[]."

Begin["`Private`"]

logfile=""

OpenLog[ filename_String, opts___?OptionQ ] := (
    logfile = OpenWrite[filename, opts];
    If[ logfile === $Failed, Return[logfile] ];
    AppendTo[$Echo, logfile];
    AppendTo[$Output, logfile];
    logfile
    )

CloseLog[ ] := (
    $Echo = Complement[$Echo, {logfile}];
    $Output = Complement[$Output, {logfile}];
    Close[logfile]
    )

End[ ]

Protect[ OpenLog, CloseLog ]

EndPackage[ ]
```

Listing 9.4–1: SessionLog.m: Logging input and output

`OpenLog[]` opens the file given as argument and, if it succeeds, appends it to both `$Echo` and `$Output`. `CloseLog[]` removes the log file from both `$Echo` and `$Output` and then closes it. If you do not close the log file before the end of the session, it will be closed by *Mathematica* as part of exit processing.

We want all input and output to go to the file session.log.

```
In[1]:= OpenLog["session.log"]

Out[1]= OutputStream[session.log, 4]
```

Here we generate some output.

```
In[2]:= Expand[ (x+y)^3 ]

         3      2          2    3
Out[2]= x  + 3 x  y + 3 x y  + y
```

And then close the log file.

```
In[3]:= CloseLog[]
Out[3]= session.log
```

After the above session, session.log contains the following text:

```
Out[1]= OutputStream["session.log", 4]
In[2]:= Expand[ (x+y)^3 ]
Out[2]= x^3 + 3*x^2*y + 3*x*y^2 + y^3
In[3]:= CloseLog[]
```

If you want the output to be rendered in output form, use

```
OpenLog[filename, FormatType -> OutputForm]
```

to open the log file.

■ 9.4.3 Advanced Topic: A Sophisticated Transcript

The logging mechanism from Section 9.4.2 prints input and output lines the way they appear in the session, including input and output prompts. The transcript code in this subsection produces a complete notebook that you can open with the frontend. The notebook will contain all input and output, formatted in `StandardForm` in individual typeset cells. The code shows how to use `$Post` and `$Epilog` to write information to a file after each evaluation. It is reproduced in Listing 9.4–3.

The command `makeTranscript`, which writes the input and output values to the log file, is assigned to `$Post`. It is therefore called once for each evaluation. At the time it is called, the input has already been assigned to `In[`n`]`. The output is passed as argument to `$Post` anyway. In this way, we have access to the current input and output to convert them to cell expressions that we can write to the output file. Because we do not wish to log the command that starts the transcript, we delay it by one evaluation. We achieve this by assigning to `$Post` a command that redefines `$Post` the next time it is called. It looks essentially like this:

```
$Post := ($Post = makeTranscript; Identity)
```

The use of `Identity` makes sure that this temporary value of `$Post` is invisible and does not disturb normal evaluation.

The command `NotebookLog[`*filename*`]` opens the file to which the notebook will be written, writes the title cell, and then sets up `$Post` in the manner just described. To close the transcript, use `NotebookLog[]`. This command closes the file and unsets `$Post`. Note that we protect `$Post` during the time that the transcript is active to prevent users from modifying it. The additional definitions for `NotebookLog[]` generate error messages if an attempt is made to open a transcript twice or to close it when it is not open. The variable `open` records whether the transcript is open. The transcript written has the following outline:

```
Notebook[{
Cell["Session Transcript", "Title"],
⋮
Cell[BoxData[...], "Input"],
Cell[BoxData[...], "Output"],
⋮
Cell["end of transcript", "SmallText"]
}]
```

The structure of notebooks is explained in Section 11.2, and the box data structures for the input and output expressions are discussed in Section 9.5.1. Here, let us concentrate on how to write out such a file. The transcript file is opened with a format type of `InputForm`. This makes it easy to write out the cell expressions using

`Write[transcript, Cell[BoxData[`*contents*`], "`*style*`"]]`

where *contents* is the formatted expression (obtained with `ToBoxes[`*expression*`]`), and where *style* is either `Input` or `Output`. The first line in the transcript is not a syntactically valid expression, so we write it out as a string, but in output form to suppress the quotation marks that input form would print. The same is done for the last line. The commas between the cells are written in a similar way, using `Write[transcript, OutputForm[","]]`.

A tricky part is the formatting of the input without evaluating it. The input expression is the value of `In[-1]` (the last value assigned to `In[]`). If we used `ToBoxes[In[-1]]`, this input would be evaluated, and `ToBoxes[Unevaluated[In[-1]]]` would format the expression `In[-1]`, rather than its value! First note that we can obtain an unevaluated form of the input using `DownValues[In][[-1]]`. The result is of the form `HoldPattern[In[`*n*`]] :>` *input*. We can use substitution and pattern matching to extract a part of an expression *expr* and insert it into another expression *new* without evaluation. The template is

`Replace[`*expr*`,` *form* `:>` *new*`]`,

where *form* is a pattern that matches the whole of *expr*. In our case we use

`Replace[DownValues[In][[-1]], (_ :> r_) :> ToBoxes[Unevaluated[r]]`.

This command starts the logging process.

```
In[1]:= NotebookLog["session.nb"]
```

As usual, we generate some sample output.

```
In[2]:= Nest[1 + 1/#&, x, 4]
```

$$\texttt{Out[2]= } 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \frac{1}{x}}}}$$

In this output, we see the short form of the result.

```
In[3]:= Short[ Expand[(x+y+1)^10], 3 ]
Out[3]//Short=

                2        3        4        5        6
 1 + 10 x + 45 x  + 120 x  + 210 x  + 252 x  + 210 x  +

               2  8       9          9    10
  <<55>> + 45 x  y  + 10 y  + 10 x y  + y
```

This will do for now.

```
In[4]:= NotebookLog[]
Out[4]= session.nb
```

Now let us look at the notebook generated. Listing 9.4–2 shows it in textual form, and Figure 9.4–1 shows the notebook as it appears in the frontend.

```
Notebook[{
Cell["Session Transcript", "Title", TextAlignment -> Center],
Cell[BoxData[\(Nest[\(\(\(1 + 1\/#1\) &\), x, 4\)]\)], "Input"],
Cell[BoxData[\(1 + 1\/\(1 + 1\/\(1 + 1\/\(1 + 1\/x\)\)\)\)], "Output"],
Cell[BoxData[TagBox[\(Expand[\(\((x + y + 1)\)\^10\)]\),
    Function[Short[Slot[1], 3]]]], "Input"],
Cell[BoxData[TagBox[\(1 + \(10\ x\) + \(45\ x\^2\) + \(120\ x\^3\) +
      \(210\ x\^4\) + \(252\ x\^5\) + \(210\ x\^6\) + \(120\ x\^7\) +
      \(45\ x\^8\) + \(\[LeftSkeleton] 48 \[RightSkeleton]\) +
      \(360\ x\ y\^7\) + \(360\ x\^2\ y\^7\) + \(120\ x\^3\ y\^7\) +
      \(45\ y\^8\) + \(90\ x\ y\^8\) + \(45\ x\^2\ y\^8\) + \(10\ y\^9\)
      + \(10\ x\ y\^9\) + y\^10\), Function[Short[Slot[1], 3]]]], "Output"],
Cell["end of transcript", "SmallText"]
}]
```

Listing 9.4–2: session.nb: A notebook generated with NotebookLog

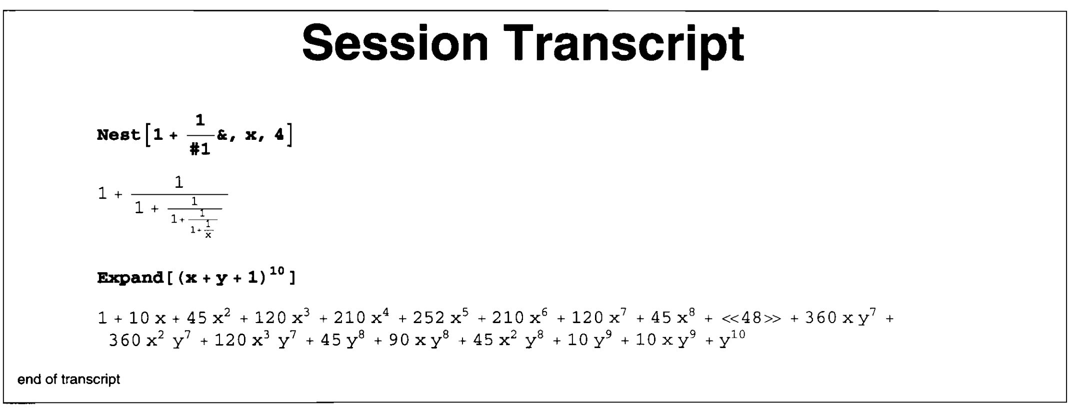


Figure 9.4–1: The notebook from Listing 9.4–2 as displayed by the frontend

The code in NotebookLog.m contains another rather subtle part: it assigns the command to close the transcript to `$Epilog`. If you forget to close the transcript with `Notebook[]`, it will be closed properly when you exit *Mathematica*. (Without this, the file would be closed, but the final lines would be missing.) Of course, when the transcript is closed, the command

to close it should be removed again from `$Epilog`. The complicated manipulations inside `closeOnExit` and `restoreExit` ensure that any other value of `$Epilog` is preserved. I leave it to you to ponder the subtleties involved.

If you want to save all the values assigned to `Out[]` during the session so far, you can simply give the command `Save["`*file*`.m", Out]`. To do this automatically before exiting *Mathematica*, you can put

`$Epilog := Save["`*file*`.m", Out]`

into your init.m file.

```
BeginPackage["ProgrammingInMathematica`NotebookLog`"]

NotebookLog::usage = "NotebookLog[\"file.nb\"] starts a transcript in notebook
    format. NotebookLog[] closes the log file."

Begin["`Private`"]

NotebookLog::notso = "Log file is not open."
NotebookLog::already = "Log file is already open."
NotebookLog::post = "Note: old value of $Post overwritten."

fileopts = {FormatType -> InputForm, PageHeight -> Infinity, PageWidth -> 78}

prolog = OutputForm["Notebook[{"]
first = Cell["Session Transcript", "Title", TextAlignment -> Center]
last = Cell["end of transcript", "SmallText"]
epilog = OutputForm["}]"]
comma = OutputForm[","]

`transcript
`open = False

NotebookLog[filename_String]/; !open := (
    transcript = OpenWrite[filename, fileopts];
    If[ transcript === $Failed, Return[transcript] ];
    If[ ValueQ[$Post], Message[NotebookLog::post] ];
    $Post := ($Post = makeTranscript; Protect[$Post]; Identity);
    closeOnExit; open = True;
    Write[transcript, prolog];
    Write[transcript, first, comma]; )

NotebookLog[filename_String]/; Message[NotebookLog::already] := Null

e:NotebookLog[]/; open := (
    Unprotect[$Post]; Unset[$Post];
    Write[transcript, last];
    Write[transcript, epilog];
    restoreExit; open = False;
    Close[transcript] )

e:NotebookLog[]/; Message[NotebookLog::notso] := Null

inCell[boxes_]  := Cell[BoxData[boxes], "Input"]
outCell[boxes_] := Cell[BoxData[boxes], "Output"]
```

```
makeTranscript =
    Function[output,
        Module[{in, out},
          in = Replace[DownValues[In][[-1]], (_ :> r_) :> ToBoxes[Unevaluated[r]]];
          Write[transcript, inCell[in], comma];
          If[ output =!= Null,
              out = ToBoxes[output];
              Write[transcript, outCell[out], comma] ]
        ];
        output ]

closeOnExit :=
    If[ ValueQ[$Epilog],
        OwnValues[$Epilog][[-1]] /.
          (l_ :> CompoundExpression[r__]|r_) :>
            ($Epilog := (r; NotebookLog[])),
        (* else *) $Epilog := NotebookLog[] ]

restoreExit := OwnValues[$Epilog][[-1]] /.
          {(l_ :> (a___; NotebookLog[]; z___)) :> ($Epilog := (a; z)),
           (l_ :> NotebookLog[]) :> Unset[$Epilog]}

End[]

Protect[ NotebookLog ]

EndPackage[]
```

Listing 9.4–3: NotebookLog.m: A sophisticated transcript

■ 9.5 Advanced Topic: Typesetting Mathematics

Version 3 introduces typeset formulae as one of its major new features. Typeset expressions are described in terms of *Mathematica* expressions themselves; therefore, they can be manipulated with programs written in *Mathematica*. The two most important manipulations are *formatting* and *interpretation*.

Formatting means to take an expression, for example, `Sum[f[i],{i, 1, n}]`, and turn it into another expression that describes how the former expression should be rendered, for example,

```
RowBox[{UnderoverscriptBox["\[Sum]", RowBox[{"i", "=", "1"}], "n"],
      RowBox[{"f", "[", "i", "]"}]}].
```

The frontend will display this box expression as $\sum_{i=1}^{n} f[i]$. Typeset expression are given in terms of boxes, objects that have a size and position on the screen.

Interpretation is the transformation in the other direction. Take a box expression, such as `RadicalBox["a", "3"]`, and infer the intended interpretation, in this case `Power[a, 1/3]`. Interpretation is a form of parsing, and as in ordinary parsing not every box expression will have a meaningful interpretation. Boxes built using the frontend's equation editor will usually be formally correct.

`ToBoxes[`*expr*`]` invokes the formatting rules and turns an expression into a typeset expression.

```
In[1]:= ToBoxes[(a + b)c]
Out[1]= RowBox[{RowBox[{(, RowBox[{a, +, b}], )}],  , c}]
```

`ToExpression[`*box*`]` is the inverse: it interprets a box expression and turns it into a *Mathematica* expression.

```
In[2]:= ToExpression[ % ]
Out[2]= (a + b) c
```

This section explains the structure of typeset expressions and shows how you can write your own formatting and interpretation rules.

■ 9.5.1 The Structure of Typeset Expressions

A typeset expression is a data structure that describes how another expression should be rendered. This description happens in terms of *boxes*. A box has contents: a sequence or list of boxes or primitive expressions, given as character strings. The type of box determines how its contents are laid out relative to each other.

The primitive elements are always given as strings, rather than as symbols. Remember, that the default output format does not show the quotes around these strings.

■ 9.5.1.1 Row Boxes and Expression Structure

The simplest box is the row box. It looks like

`RowBox[{`e_1`, `e_2`, ...}, `*options...*`]`.

When rendered, it lays out its contents next to each other. The spacing between the elements is determined according to typesetting rules similar to the ones used in TeX. It is important to understand that the contents cannot be any sequence of expressions. They should be syntactically correct and nested according to the structure of the expression that is typeset.

For example, the typeset expression

```
RowBox[{RowBox[{"a", "+", "b"}], "c"}]        (9.5–1)
```

is interpreted as $(a + b)c$, even though no parentheses are visible (if you evaluate such an expression, *Mathematica* will insert the missing parentheses in its output). The box

```
RowBox[{"a", "+", RowBox[{"b", " ", "c"}]}],        (9.5–2)
```

however, is interpreted as $a + (bc)$.

When you enter a typeset expression with the frontend, the appropriate nesting is done automatically. The nested boxes in 9.5–2, for example, are constructed automatically when you type the characters `a`, `+`, `b`, SPACE, and `c` in sequence.

You can bypass the automatic nesting by switching a cell into Format ▷ Show Expression mode to see the box structure. You can edit this structure and then switch back to display mode to see the effects of your changes. Working in this mode is an easy way to experiment with the structure of typesetting expressions, which is sometimes needed for figuring out the correct formatting and interpretation rules that will be discussed in the following subsections. The boxes in 9.5–1 cannot be constructed by entering single keystrokes, only in Show Expression mode. Figure 9.5–1 shows these experiments in a notebook.

■ 9.5.1.2 Types of Boxes

Boxes can be nested. The contents of a box are either another box or a string. Table 9.5–1 lists the types of boxes that *Mathematica* understands, together with their meaning and display form.

Boxes can have options; for example, `GridBox[{...}, ColumnAlignments->Left]` causes the columns of a grid to be aligned left instead of centered (the default). The details are given in Subsection 2.8.10 of the *Mathematica* book.

■ 9.5.2 Formatting Rules

Prior to Version 3, you defined print formats with rules for `Format[`*pattern*`, `*formattype*`]`, as explained in Section 9.1.2. Internally, such rules are now translated into equivalent rules for the new `MakeBoxes[]` primitive operation.

Row Boxes

■ Typing a + b _ c

a + b c

Underlying boxes, obtained with the Show Expression menu item. Note the nested boxes.

```
Cell[BoxData[
    RowBox[{"a", "+",
      RowBox[{"b", " ", "c"}]}]], "Input"]
```

■ Constructing the Boxes Yourself

In Show Expression mode, you can enter boxes yourself.

```
Cell[BoxData[
    RowBox[{
      RowBox[{"a", "+", "b"}], "c"}]], "Input"]
```

Formatted and then evaluated. The input is misleading, because priorities are not what they seem.

a + b c

(a + b) c

The unformatted form of the result. Note the additional parentheses.

```
Cell[BoxData[
    RowBox[{
      RowBox[{"(",
        RowBox[{"a", "+", "b"}], ")"}], " ", "c"}]], "Output"]
```

Figure 9.5–1: Row boxes, formatted, and unformatted expressions

`RowBox[{`b_1`, `b_2`, ...}]`	row of elements $b_1 b_2 \ldots$
`SubscriptBox[`a`, `b`]`	subscript a_b
`SuperscriptBox[`a`, `b`]`	superscript a^b
`SubsuperscriptBox[`a`, `b`, `c`]`	subscript and superscript a_b^c
`UnderscriptBox[`a`, `b`]`	underscript $\underset{b}{a}$
`OverscriptBox[`a`, `b`]`	overscript $\overset{b}{a}$
`UnderoverscriptBox[`a`, `b`, `c`]`	underscript and overscript $\overset{c}{\underset{b}{a}}$
`FractionBox[`n`, `d`]`	fraction $\frac{n}{d}$
`SqrtBox[`a`]`	square root $\sqrt{a}$
`RadicalBox[`a`, `r`]`	r^{th} root $\sqrt[r]{a}$
`GridBox[{{`a_{11}`, `a_{12}`, ...}, {`a_{21}`, ...}, ...}]`	grid or matrix $\begin{matrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \ddots \end{matrix}$
`Rowbox[{"(", GridBox[{...}], ")"}]`	matrix $\begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$
`FrameBox[`*box*`]`	frame around $\boxed{box}$
`StyleBox[`*box*`, "`*style*`"]`	render *box* in specified style
`StyleBox[`*box*`, `*options...*`]`	render *box* with specified formatting options
`ButtonBox[`*box*`]`	turn *box* into clickable button

Table 9.5–1: Boxes and their rendering

Here is a formatting rule for the function `f` that prints the first argument as a subscript.

```
In[1]:= Format[b:f[_, _]] := Subscripted[b, 1]
```

It has been turned into an upvalue for `MakeBoxes`.

```
In[2]:= ??f

Global`f

MakeBoxes[b:f[_, _], FormatType_] ^:=
  Format[Subscripted[b, 1], FormatType]

Format[b:f[_, _]] := Subscripted[b, 1]
```

It works the way it always did.

```
In[3]:= f[1, x]

Out[3]= f [x]
         1
```

When an expression is formatted, either to be displayed by the frontend, or because you asked for its box form with `ToBoxes[`*expr*`]`, *Mathematica* takes care of all the difficult details of protecting the expression from evaluation and calls `MakeBoxes[`*expr*`, `*formattype*`]` on the parts of the expression that need formatting. The format type *formattype* is one of `StandardForm` (the default) or `TraditionalForm`.

To define a format for an expression *h*`[`*elements*`]` you give a rule for

h`/:MakeBoxes[`*h*`[`*elements*`], form_] := ...`

if your rule should apply to all format types, or

h`/:MakeBoxes[`*h*`[`*elements*`], form:TraditionalForm] := ...`

if it should apply only to traditional form, for example. In either case, the symbol `form` stands for the current format type on the right side of the rules.

Your rule should build an appropriate box and invoke `MakeBoxes[`*expr*`, form]` recursively on the parts of *h*`[`*elements*`]` that appear in your boxes. (Here we use the symbol `form` mentioned in the preceding paragraph.)

To find out how the boxes must look, you can consult the frontend (look at the box expression with the Format ▷ Show Expression menu) or you can invoke `ToBoxes[`*expr*`]` on similar expressions. For an example, let us define an appropriate traditional form for the Struve functions introduced in Section 8.4.

■ 9.5.2.1 Formatting Struve Functions

Struve functions are similar to Bessel functions. They have two arguments; the first argument is a kind of index. The traditional rendering of `StruveH[`*n*`, `*z*`]` is $H_n(z)$, which is similar to the traditional form $J_n(z)$ of a Bessel function, which is already built into *Mathematica*.

Here we see how *Mathematica* renders $J_n(z)$ in traditional form.

```
In[4]:= ToBoxes[BesselJ[n, z], TraditionalForm]

Out[4]= FormBox[RowBox[{SubscriptBox[J, n], (, z, )}],
  TraditionalForm]
```

The `FormBox[]` wrapper records the fact that this box is in traditional form. We need only the box itself.

```
In[5]:= %[[1]]
Out[5]= RowBox[{SubscriptBox[J, n], (, z, )}]
```

As explained, the symbols in the boxes are, in fact, strings. Here you can see the quotes around them.

```
In[6]:= FullForm[ % ]
Out[6]//FullForm=
 RowBox[List[SubscriptBox["J", "n"], "(", "z", ")"]]
```

Another rendering that you will often encounter in the frontend is the linear syntax, in which a row box `RowBox[{`e_1`, `e_2`, ..., `e_n`}]` is written as `\(`e_1 e_2 ... e_n`\)`.

```
In[7]:= InputForm[ % ]
Out[7]//InputForm= \(J\_n(z)\)
```

Now we know that the expression `StruveH[`n`, `z`]` should be formatted as

`RowBox[{SubscriptBox["H", `n`], "(", `z`, ")"}].`

However, the two variable parts n and z need to be formatted as well (they can be complicated expressions themselves). The correct procedure is to invoke `ToBoxes[]` recursively on all such parts. Here is the formatting rule for Struve functions in traditional form.

```
StruveH/:
MakeBoxes[StruveH[n_, z_], form:TraditionalForm] :=
    RowBox[{SubscriptBox["H", MakeBoxes[n, form]], "(", MakeBoxes[z, form], ")"}]
```

Part of Struve.m: Formatting

Struve functions are now formatted like Bessel functions, and their arguments are recursively formatted. What you see here is an approximation of the nicely typeset output $H_1(x^2)$ you would get in the frontend.

```
In[8]:= StruveH[1, x^2] // TraditionalForm
                               2
Out[8]//TraditionalForm= H (x )
                          1
```

■ 9.5.3 Interpretation Rules

Interpretation rules convert boxes into ordinary *Mathematica* expressions. The function `ToExpression[`*box*`, `*formattype*`]` invokes `MakeExpression[`*box*`, `*formattype*`]` on the subparts and assembles the results into an expression. It is careful not to evaluate the intermediate expressions, essentially keeping the expressions inside `HoldAllComplete` at all times.

Fortunately, if you want to give interpretation rules for a notation you made up yourself, you need not parse the expression completely; all you need to do is to turn it into a box form that *Mathematica* already knows how to handle. Interpretation rules, therefore, are often iterative, rather than recursive.

Let us continue the Struve function example and give interpretation rules that turn the traditional typeset form from Section 9.5.2.1 back into an expression.

■ 9.5.3.1 Interpretation of Struve Functions

From Section 9.5.2.1 we know that the traditional form $H_n(z)$ is represented by the box structure `RowBox[{SubscriptBox["H", n], "(", z, ")"}]`. Our rule must match such a box expression and turn it into something *Mathematica* can recognize. The standard form is $H[n, z]$; we can ask *Mathematica* to give us its box representation.

Here is the box form for $H[n, z]$.

```
In[9]:= ToBoxes[H[n, z]]
Out[9]= RowBox[{H, [, RowBox[{n, ,, z}], ]}]
```

It becomes clearer with quotes around the strings, especially regarding the embedded `","`.

```
In[10]:= FullForm[ % ]
Out[10]//FullForm=
 RowBox[List["H", "[", RowBox[List["n", ",", "z"]], "]"]]
```

Here is the rule that turns the box form of $H_n(z)$ into the box form of $H[n, z]$:

```
MakeExpression[ RowBox[{SubscriptBox["H", n_], "(", z_, ")"}],
                form:TraditionalForm ] :=
  MakeExpression[ RowBox[{"StruveH", "[", RowBox[{n, ",", z}], "]"}], form ]
```

Part of Struve.m: Interpretation

Note that you do not need to use recursion on n and z. These variables are still boxes that will be converted by the built-in rules.

With this rule, the traditional form expression $H_1(\sqrt{2})$, representing a Struve function, is correctly recognized. The notation `\!\(TraditionalForm\` ... \)` builds `FormBox[..., TraditionalForm]`, to tell the parser that this is in traditional form; it is needed here, because the default interpretation format type is `StandardForm`.

```
In[11]:= \!\(TraditionalForm\` H\_1(\@2) \)
Out[11]= StruveH[1, Sqrt[2]]
```

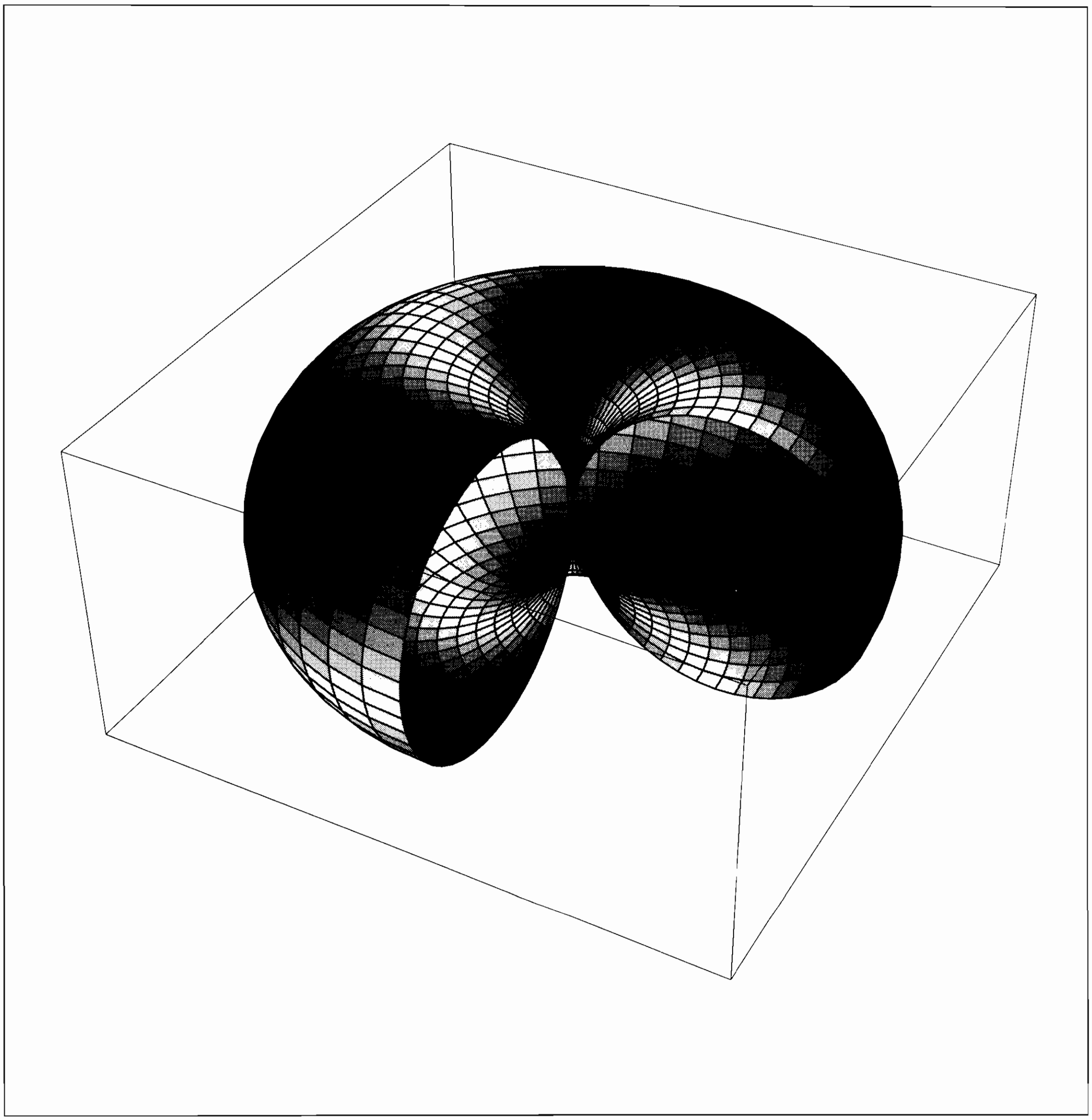
When designing interpretation rules you will usually turn your boxes into the box representation of a normal expression in the form h`[`e_1`, `e_2`, ..., `e_n`]`, because *Mathematica* knows very well how to parse such a box form. (This normal form is valid in both `TraditionalForm` and `StandardForm`.) Note that the elements between the square brackets are in a subordinate row box that uses commas as operators; that is, the box form looks like

`RowBox[{`h`, "[", RowBox[{`e_1`, ",", `e_2`, ",", ..., ",", `e_n`}], "]"}]`.

Only in the case $n = 0$ or $n = 1$ is there no inside row box.

Chapter 10

Graphics Programming



Several utilities for graphics and animation are collected in this chapter. The utilities for parametric plots and for coloring graphs in Section 1 have been developed to make the production of the cover picture for this book easier. They turned out useful in many other applications as well.

The topic of Section 2 is *animation*. We developed a way of looking at the frames of an animation in a static medium, such as this book.

The last section discusses the code used to produce the chapter-opener and the cover pictures.

About the illustration overleaf:

This shape is similar to the cover picture of the second edition. The torus is shaded in a diagonal pattern, different on the inside and the outside.

```
SphericalPlot3D[ {Sin[theta],
    FaceForm[GrayLevel[0.05 + 0.9 Sin[2theta + phi]^2],
            GrayLevel[0.05 + 0.9 Sin[2theta - phi]^2]]},
   {theta, 0, Pi, Pi/48}, {phi, 0, 3Pi/2, Pi/24}, Lighting->False ]
```

10.1 Graphics Packages

10.1.1 More 3D Parametric Plotting

The command `Plot3D[]` allows you to define the color or gray level of the surface plotted under program control. The form is `Plot3D[{`z`,` *style*`}, ...]`, where *style* can be `RGBColor[]` or `GrayLevel[]`, for example. We can do the same thing for parametric plots. Normally a parametric plot is given as

`ParametricPlot3D[{`x`,` y`,` z`}, {`u`,` u_0`,` u_1`}, {`v`,` v_0`,` v_1`}, ...]`

with x, y, and z being functions of u and v. If we want to specify the color of the surface patches as a function of u and v, the syntax is

`ParametricPlot3D[{`x`,` y`,` z`,` *style*`}, {`u`,` u_0`,` u_1`}, {`v`,` v_0`,` v_1`}, ...]`.

The command `ParametricPlot3D[]` uses the option `PlotPoints` to specify the number of lines to draw in each direction. This is consistent with `Plot[]` and `Plot3D[]`. In version 1.2 of *Mathematica*, `ParametricPlot3D[]` was not built-in, but was provided in a package. This package allowed the number of lines to be given as an increment in the two iterators for u and v in the form

`ParametricPlot3D[{`x`,` y`,` z`}, {`u`,` u_0`,` u_1`,` d_u`}, {`v`,` v_0`,` v_1`,` d_v`}, ...]`.

Because we wanted to keep this functionality, we developed a (new) package Graphics/ParametricPlot3D.m with additional rules for the now built-in `ParametricPlot3D[]`. It contains also the commands `SphericalPlot3D[]` and `CylindricalPlot3D[]`, as well as some commands for drawing curves in three dimensions.

The additional rules for `ParametricPlot3D[]` match whenever at least one of the iterators has an increment specified. It then computes the appropriate value for the `PlotPoints` option and calls the built-in code. The technique for handling the optional increments is the one explained in Section 3.1.3.

```
ParametricPlot3D[f_, {u_, u0_, u1_, du_:Automatic}, {v_, v0_, v1_, dv_:Automatic},
                 opts___?OptionQ ] :=
    Module[{plotpoints},
        plotpoints = PlotPoints /. {opts} /. Options[ParametricPlot3D];
        If[ Head[plotpoints] =!= List, plotpoints = {plotpoints, plotpoints} ];
        If[ du =!= Automatic, plotpoints[[1]] = Round[(u1-u0)/du] + 1 ];
        If[ dv =!= Automatic, plotpoints[[2]] = Round[(v1-v0)/dv] + 1 ];
        ParametricPlot3D[f, {u, u0, u1}, {v, v0, v1}, PlotPoints->plotpoints, opts]
    ] /; du =!= Automatic || dv =!= Automatic
```

Part of ParametricPlot3D.m: Overloading `ParametricPlot3D[]`.

The value of PlotPoints can be either a single number, valid for both iterators, or a list of two values, one for each iterator. Therefore, we check whether the value extracted from the options is a list. If it is not a list, we reset it to a list with both elements being the same. If du is present (if it is not the symbol Automatic) we compute the number of plot points and reset the first element of the value of plotpoints. We do the same for dv and then call ParametricPlot3D[] again, this time without the increments du and dv, but with the new value plotpoints for the PlotPoints option. Our rule no longer matches this case and the built-in code takes over.

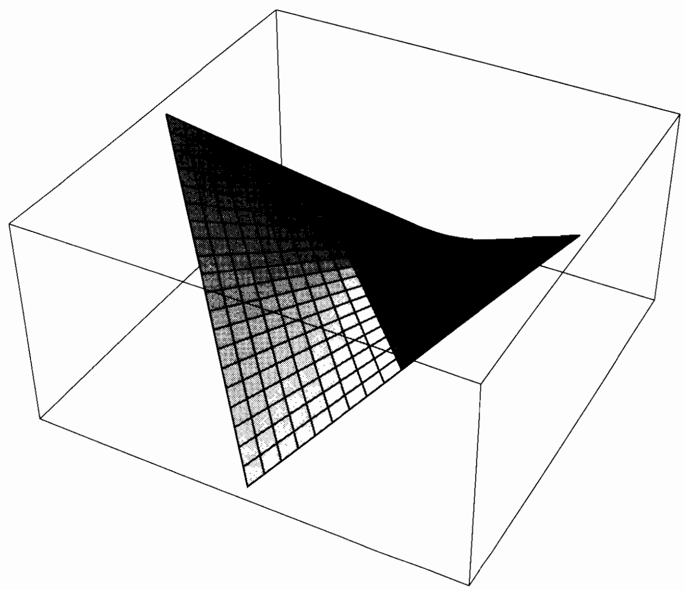
The package is in the Graphics directory of the standard *Mathematica* distribution.

```
In[1]:= Needs["Graphics`ParametricPlot3D`"]
```

We get 20 lines for u and the default number of lines for v. This picture shows a Cartesian parameterization of the saddle surface; see also page 159.

To see the colors or gray levels we have to turn off lighting of the surface.

```
In[2]:= ParametricPlot3D[ {u - v, u + v, u v,
           FaceForm[GrayLevel[ArcTan[u, v]/2/Pi + 0.5]]},
          {u, -1, 1, 2/20}, {v, -1, 1},
          Lighting -> False,
          ViewPoint -> {-2.4, -1.3, 2.0}
        ];
```



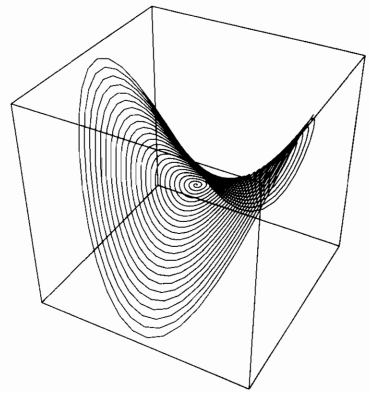
If only one iterator is given in ParametricPlot3D, a curve in space is drawn. It is a simple matter to compute the appropriate value of PlotPoints from the increment given in the iterator.

```
ParametricPlot3D[ fun_, {u_, u0_, u1_, du_}, opts___?OptionQ ] :=
    ParametricPlot3D[ fun, {u, u0, u1}, PlotPoints -> Round[(u1-u0)/du] + 1, opts]
```

Part of ParametricPlot3D.m: A rule for curves in space

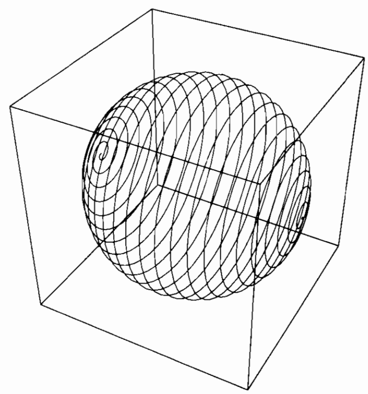
This curve lies on the saddle surface.

```
In[3]:= ParametricPlot3D[
          {t Cos[48Pi t], t Sin[48Pi t], t^2 Cos[96Pi t]},
          {t, 0, 1, 0.001} ];
```



If the iterator does not contain an increment, to built-in code is used as if our package had not been loaded. Here is a line that spirals around a sphere.

```
In[4]:= ParametricPlot3D[
          {Cos[Pi t], Sin[Pi t] Sin[48Pi t],
           Sin[Pi t] Cos[48Pi t]},
          {t, 0, 1}, PlotPoints->1000 ];
```



■ 10.1.2 Plot Utilities

The commands `RGBColor[]` and `GrayLevel[]` are rather finicky about their arguments. They do not like numbers outside the interval `{0, 1}`. We want to define some utilities for specifying colors and gray levels that are easier to use.

For the cover picture of the second edition we used the function `ColorCircle[]`, which translates its argument into a hue value and takes it modulo 2π. The argument of the built-in function `Hue[]` is taken modulo 1. Therefore, we divide by 2π. The optional second argument specifies the overall brightness of the colors.

The second set of commands transforms the phase angle or argument of a complex number into a gray level or color value. Because the phase angle ranges from $-\pi$ to π

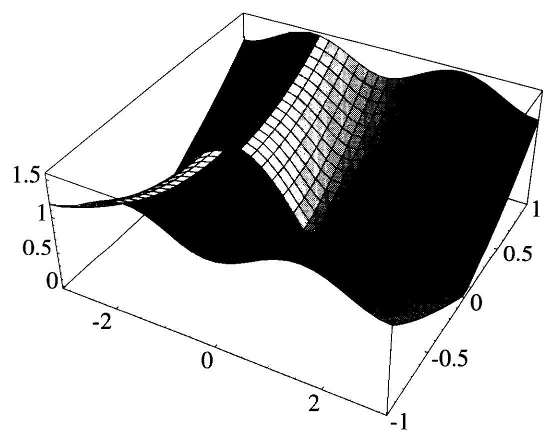
(once around the circle) it is natural to let it determine the hue of a color. We use the function ColorCircle[] that we just defined. ArgShade[] uses gray levels instead of colors. Because the phase angle of 0 is undefined we need a special rule for this case. Listing 10.1–1 shows the complete package ArgColors.m.

```
BeginPackage["Graphics`ArgColors`"]

ArgColor::usage = "ArgColor[z] gives a color value whose hue is proportional
    to the argument of the complex number z."
ArgShade::usage = "ArgShade[z] gives a gray level proportional
    to the argument of the complex number z."

ColorCircle::usage = "ColorCircle[r, (light:1)] gives a color value whose hue
    is proportional to r (mod 2Pi) with lightness light."

Begin["`Private`"]

ArgColor[z_] /; z == 0.0 := Hue[0, 0, 1]
ArgColor[z_?NumericQ] := ColorCircle[ Arg[z] ]

ArgShade[z_] /; z == 0.0 := GrayLevel[1]
ArgShade[z_?NumericQ] := GrayLevel[ N[(Pi + Arg[z])/(2Pi)] ]

ColorCircle[r_?NumericQ, light_:1] := Hue[ N[r/(2Pi)], 1, light ]

End[]

Protect[ArgColor, ArgShade, ColorCircle]

EndPackage[]
```

Listing 10.1–1: ArgColors.m: Colors for complex numbers.

The function value is the absolute value of the sine function; the gray level is determined by the phase angle of the sine function. Using ArgColor[] on a color display instead of ArgShade[], the sharp jump from black to white does not occur because the colors close up nicely on the color circle. One of the deep mysteries of *Mathematica* is why user-defined shades in Plot3D are visible only with the default setting Lighting->True (compare with the parametric plot on page 274).

```
In[1]:= Plot3D[{Abs[Sin[x + I y]], ArgShade[Sin[x + I y]]},
              {x, -Pi, Pi}, {y, -1, 1}, PlotPoints->30 ];
```



As an aside, note that in the previous example the Sin[] function is computed *twice* for each point. You can save much time by using the following command instead:

```
Plot3D[ {Abs[z = Sin[x + I y]], ArgShade[z]}, ...].
```

■ 10.2 Animated Graphics

Animation allows time-evolving phenomena to be viewed with comparatively small hardware requirements. A number of images are displayed in rapid succession, giving the impression of smoothly moving objects. It is quite easy to produce the individual frames. A simple loop with a graphic command is sufficient. This section describes how animations are displayed and then treats the display of animations in a static medium, such as a book.

■ 10.2.1 How Animation Works

The rendering of an animation is hardware dependent. The animation functions in the package Graphics/Animation.m are therefore written in terms of two auxiliary functions, `RasterFunction[]` and `AnimationFunction[]`. These functions are defined in the device-dependent graphics initialization files, such as PSDirect.m (for use with a notebook frontend) or Motif.m (for the X Window System). The defaults are assigned to the global variables `$RasterFunction` and `$AnimationFunction`.

The function `Animate[`*graph*`,` *iterator*`,` *options*`...]` performs the graphic command *graph* iterating over the given iterator to produce the individual frames. For each frame it calls `Pixelize[`*graph*`,` *raster*`,` *options*`...]`, which in turn uses `Show[]` to render the graphics command with *raster* as the value of `DisplayFunction`. The results are collected in a list. At the end this list is passed to the animation function. `ShowAnimation[]` animates a list of existing graphics. Listing 10.2–1 shows the definitions of these functions, taken from Graphics/Animation.m. You should recognize many of the techniques for dealing with options and defaults from Chapter 3.

The `Block[]` statement inside `Animate[]` temporarily changes the default value of `$DisplayFunction`. This suppresses the rendering of intermediate frames in the case where the graphics were generated by `Plot[]` and similar functions that display their results by default. The option `Frames` gives the default number of frames to draw. `Closed->True` assumes that the last frame is identical to the first one. It is therefore not rendered, giving a smoother animation.

The defaults for the raster and animation functions work together with the default display function. They simply write the graphics in sequence to the `$Display` channel. The auxiliary function `DisplayAnimation[`*display*`, {`*graphics*`...}]` invokes `Display[]` on all graphics in the given list. The default raster function is simply the identity; the default animation function is `DisplayAnimation[$Display, #]&` (in analogy to the default display function, which is `Display[$Display, #]&`).

The graphics initialization files should override these defaults if necessary. The values used for the X window system, for example, are shown in Listing 10.2–2. The values appropriate for the frontend are discussed in Section 11.3.4.

```
Options[ ShowAnimation ] = {
    RasterFunction :> System`$RasterFunction,
    AnimationFunction :> System`$AnimationFunction }
Options[ Animate ] = { Frames -> 24, Closed -> False }

(* defaults for $RasterFunction and $AnimationFunction *)

If[ !ValueQ[System`$RasterFunction], $RasterFunction = Identity ]
If[ !ValueQ[System`$AnimationFunction],
    $AnimationFunction = DisplayAnimation[$Display, #]& ]

(* work on multiple output channels sequentially *)

DisplayAnimation[disp_List, pics_] := DisplayAnimation[#, pics]& /@ disp
DisplayAnimation[display_, pics_] := CallAnimation[display, pics]

(* open file, if it is a string that does not refer to a stream *)

CallAnimation[display_String, pics_] :=
    Module[{stream, res, open},
        stream = Streams[display]; open = Length[stream] > 0;
        If[!open, stream = OpenWrite[display], stream=stream[[1]]];
        If[ stream === $Failed, Return[stream]];
        res = CallAnimation[stream, pics];
        If[!open, Close[stream]];
        res
    ]

CallAnimation[display_, pics_] := Display[display, #]& /@ pics

Pixelize[ go_, RasterFunction_, opts___ ] :=
    Module[ {gtype = Head[go]},
        While[ gtype === List && Length[gtype] > 0, gtype = Head[gtype] ];
        Show[ go, DisplayFunction -> RasterFunction, FilterOptions[gtype, opts] ]
    ]

ShowAnimation[ gl_List, opts___ ] :=
    Module[{res, raster, animation},
        raster = RasterFunction /. {opts} /. Options[ShowAnimation];
        animation = AnimationFunction /. {opts} /. Options[ShowAnimation];
        res = Pixelize[#, raster, opts]& /@ gl;
        animation[ res ]
    ]

Attributes[Animate] = {HoldFirst};
Animate[ function_, {t_, t0_, t1_, dt_:Automatic}, opts___ ] :=
    Module[{res, raster, animation, ndt = dt, closed, nt1 = t1, frames},
        closed = Closed /. {opts} /. Options[Animate];
        raster = RasterFunction /. {opts} /. Options[ShowAnimation];
        animation = AnimationFunction /. {opts} /. Options[ShowAnimation];
        If[ dt === Automatic,
            frames = Frames-1 /. {opts} /. Options[Animate];
            If[ closed, frames++ ]; ndt = (t1 - t0)/frames ];
        If[ closed, nt1 -= ndt];
        Block[{$DisplayFunction = Identity, $SoundDisplayFunction = Identity},
          res = Table[Pixelize[function, raster, opts], {t, t0, nt1, ndt}] ];
        animation[ res ]
    ]
```

Listing 10.2–1: Part of Graphics/Animation.m: Functions for animated graphics

```
(* Animation: all frames in one file, then animate *)

`$RasterFunction = Identity;
`$AnimationFunction = $MotifAnimationFunction;

Begin[ "`Private`"]

$MotifAnimationFunction =
     Module[{file = OpenTemporary[]},
        System`DisplayAnimation[file, #];
        file = Close[file];
        $DisplayLinkCommands[file, $AnimationDisplayTitle];
        #
    ]&

End[]
```

Listing 10.2–2: Part of X11.m: Animation settings

■ 10.2.2 A Static View of Animated Graphics

In a printed medium, such as this book, it is unfortunately not possible to include animated graphics. All we can do is capture the individual frames of the animation and display them in a graphics array. The package FlipBookAnimation.m contains definitions for the animation primitives `RasterFunction[]` and `AnimationFunction[]`, mentioned in the preceding section, which achieve this effect. Understanding how these work should allow you to write your own versions of these animation functions for particular circumstances not covered by the versions distributed with *Mathematica*.

The raster function is simply the identity, because we want only to collect the individual graphs in a list, not render them. At the end, `$AnimationFunction` is called with the list of the graphs as argument. It puts them into a matrix of graphs by breaking the list into rows of equal length. By choosing the length of the rows to be near the square root of the number of graphs, we get a matrix which is nearly square. If the number of graphs has no suitable divisors, this is not possible. In this case, we generate a partially filled last row. `GraphicsArray[]` can cope with an incomplete last row. To determine the length of the rows we first find all divisors of the number l of graphs, then select the smallest divisor r that is larger than $\sqrt{l}$. If r is not too large, we use this value; otherwise, we choose the smallest integer larger than $\sqrt{l}$. If there are only one or two graphs, we put them into a single row.

There is a global variable, `Graphics`Animation`$Columns`, that can be used to force a particular number of columns. It is useful for producing a series of animations with identical numbers of columns.

The code is shown in Listing 10.2–3. The file is part of *Mathematica*; it can be found in the directory SystemFiles/Graphics/Packages. This directory is normally on the search path `$Path`.

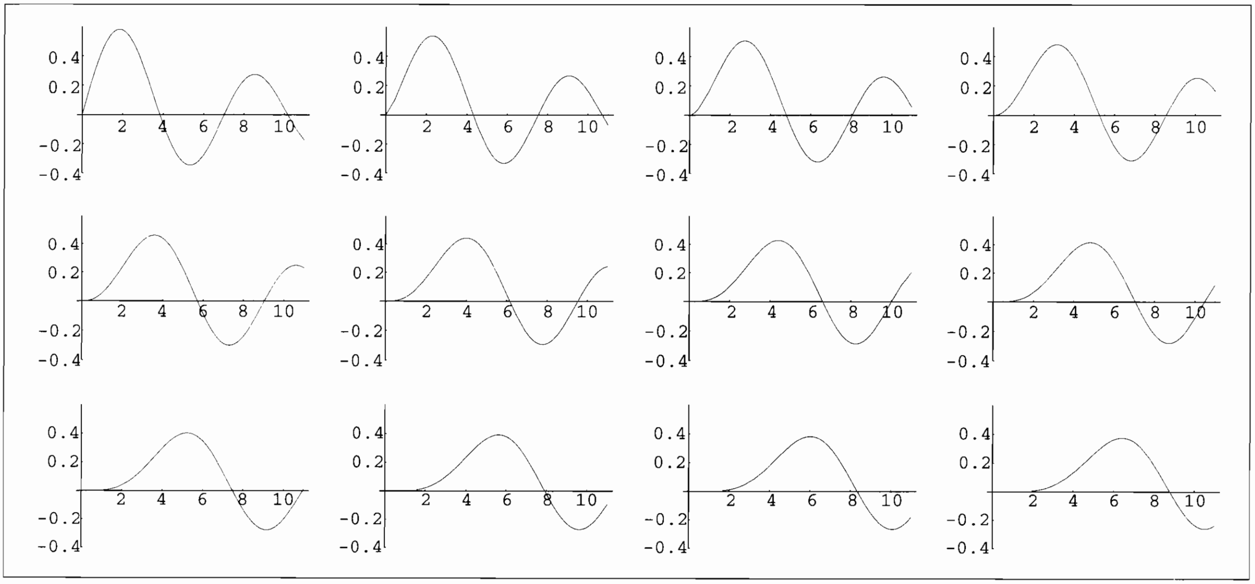
To use flipbook animation, you need to read in both the animation package and FlipBookAnimation.m. Then, you can issue animation commands as usual; the results will be displayed in a picture, either in your notebook or somewhere on your screen.

This sets up the animation functions and reads the definitions for static animations.

```
In[1]:= Needs["Graphics`Animation`"];\
        << FlipBookAnimation.m
```

We generate 12 frames of Bessel functions J_t, varying their index t from 1 to 5. The 12 graphs are rendered in a 3 by 4 matrix.

```
In[2]:= Animate[ Plot[ BesselJ[t, x],
                       {x, 0, 11},
                       PlotRange->{-0.4, 0.6},
                       DefaultFont -> {"Courier", 6} ],
                 {t, 1, 5},
                 Frames -> 12 ];
```



```
Graphics`Animation`$Columns::usage =  "Graphics`Animation`$Columns specifies
    the number of columns in the array of animation frames."

Begin["System`"]

$RasterFunction = Identity
$AnimationFunction = Graphics`Animation`MakeGraphicsArray

If[ !ValueQ[Graphics`Animation`$Columns],
    Graphics`Animation`$Columns = Automatic ]

Begin["Graphics`Animation`Private`"]

Graphics`Animation`MakeGraphicsArray[pics_] :=
    Module[{l = Length[pics], r, row, div, picts},
        If[ l > 1,
            div = Divisors[l];
            r = First[ Select[div, # >= Sqrt[l]&] ];
          , (* else no divisors *)
            r = l
        ];
        Which[
            IntegerQ[Graphics`Animation`$Columns] &&
            1 <= Graphics`Animation`$Columns,
                row = Graphics`Animation`$Columns, (* preset *)
            l <= 2, row = l,                  (* trivial *)
            r < 1.4 Sqrt[l], row = r,         (* can divide exactly *)
            True,   row = Ceiling[Sqrt[l]]  (* cannot partition exactly *)
        ];
        If[ l < row, (* fill in rest with dummies *)
            picts = Join[pics, Table[Graphics[{}], {row-l}]];
          , (* else partition into rows *)
            picts = Partition[pics, row];
            If[ Mod[l, row] > 0,    (* leftovers *)
                AppendTo[picts, Take[pics, -Mod[l, row]]]
            ];
        ];
        Show[GraphicsArray[picts], DisplayFunction -> $DisplayFunction]
    ]

End[] (* Graphics`Animation`Private` *)

End[] (* System` *)
```

Listing 10.2–3: FlipBookAnimation.m: Putting animation frames into a single picture

■ 10.3 The Chapter Pictures

The package BookPictures.m (Listing 10.3–1) and the notebook BookPictures.nb contain the code to produce the pictures on the chapter title pages. The picture for Chapter n is produced by evaluating the symbol `chapter`n. Further symbols are `appendix`a and `index`. The symbols `cover`n, for $n = 1, 2, 3$, give the cover images for the three editions of this book. The first time a symbol is evaluated, the graphic is computed and displayed, which may take some time. The graphic is then remembered and can be rendered again without the need for recomputation. This technique was explained in Section 5.5.3.

```
Needs["Graphics`ParametricPlot3D`"]
Needs["Graphics`Shapes`"]
Needs["Graphics`ArgColors`"]
Needs["Graphics`Colors`"]

Needs["ProgrammingInMathematica`ComplexMap`"]
Needs["ProgrammingInMathematica`RungeKutta`"]
Needs["ProgrammingInMathematica`RandomWalk`"]

Needs["ProgrammingInMathematica`ChaosGame`"]

SetOptions[Graphics3D, PlotRange -> All]
SetOptions[ParametricPlot3D, PlotRange -> All]

(* Moebius transform *)

chapter1 := chapter1 =
PolarMap[ (2# - I)/(# - 1 + 0.1I)&, {0.001, 5}, {0, 2Pi},
          Frame -> True, Lines -> {20, 36}, PlotPoints -> 40 ]

(* Minimal surface *)

chapter2 := chapter2 =
ParametricPlot3D[{r*Cos[phi] - (r^2*Cos[2*phi])/2,
                  -(r*Sin[phi]) - (r^2*Sin[2*phi])/2,
                  (4*r^(3/2)*Cos[(3*phi)/2])/3},
                 {r, 0.0001, 1}, {phi, 0, 4Pi}, PlotPoints -> {8, 60}]

(* rotationally symmetric parametric surface *)

chapter3 := chapter3 =
ParametricPlot3D[
    {r Cos[Cos[r]] Cos[psi], r Cos[Cos[r]] Sin[psi], r Sin[Cos[r]]},
    {r, 0.001, 9Pi/2}, {psi, 0, 3Pi/2}, PlotPoints -> {72, 30}]

(* J. Waldvogel's Christmas picture *)

chapter4 := chapter4 =
  With[{c = Pi(1 + Sqrt[5])/2.0, x = Range[50000]},
      ListPlot[{Re[#], Im[#]}& /@ FoldList[Plus, 0, Exp[I c x^2]],
               PlotJoined -> True, AspectRatio -> Automatic, Axes -> None]
  ]

(* Sphere with random holes *)

chapter5 := chapter5 =
Show[ Graphics3D[Select[Sphere[1, 72, 54], Random[]>0.5&]] ]
```

```
(* Saddle surface *)

chapter6 := chapter6 =
CylindricalPlot3D[r^2 Cos[2 phi], {r, 0, 1/2, 1/20}, {phi, 0, 2Pi, 2Pi/36}]

(* Van-der-Pol equation *)

chapter7 := chapter7 =
Module[{vdp, eps = 1.5, x, xdot},
  vdp = RKSolve[{xdot, eps(1 - x^2)xdot - x}, {x, xdot}, #, {5Pi, 0.05}]&
        /@ {{0.1,0}, {-0.1,0}, {2,-2}, {-2,2}};
  vdp = ListPlot[#, PlotJoined -> True, DisplayFunction -> Identity]& /@ vdp;
  Show[ vdp, AspectRatio -> Automatic, DisplayFunction -> $DisplayFunction ]
]

(* Fractal tile *)

hexarotation = ArcSin[Sqrt[3/7]/2];
mids = Solve[x(x^6-1) == 0];
r7 = translation[{Re[#], Im[#]}]& /@ (x /. mids);
sr = Composition[ scale[1/Sqrt[7]], rotation[hexarotation] ];
rmaps = Composition[#, sr]& /@ r7;
hexatile = IFS[ N[rmaps] ]

chapter8 := chapter8 =
    Module[{pts, gr},
        pts = Flatten[ Nest[hexatile, Point[{0,0}], 6] ];
        gr = Graphics[{PointSize[0.0015], pts}];
        Show[gr, AspectRatio -> Automatic, PlotRange -> All]
    ]

(* Fourier approximations of saw-tooth *)

l5 = Table[ Sum[Sin[i x]/i, {i, n}], {n, 10} ];

chapter9 := chapter9 =
Plot[ Evaluate[l5], {x, -0.3, 2Pi+0.3} ]

(* spiral with varying radius *)

chapter10 := chapter10 =
ParametricPlot3D[{r (1 + phi/2) Cos[phi], r (1 + phi/2) Sin[phi], -phi/2},
          {r, 0.1, 1.1, 0.125}, {phi, 0, 11Pi/2, Pi/16}]

(* diagonally shaded surface *)

chapter11 := chapter11 =
SphericalPlot3D[{Sin[theta],
                 FaceForm[GrayLevel[0.05 + 0.9 Sin[2theta + phi]^2],
                          GrayLevel[0.05 + 0.9 Sin[2theta - phi]^2]]},
                {theta, 0, Pi, Pi/48}, {phi, 0, 3Pi/2, Pi/24}, Lighting->False ]

(* Barnsley's Fern *)

bf1 = AffineMap[ -2.5 Degree, -2.5 Degree, 0.85, 0.85, 0, 1.6];
bf2 = AffineMap[ 49. Degree, 49. Degree, 0.3, 0.34, 0, 1.6];
bf3 = AffineMap[ 120. Degree, -50. Degree, 0.3, 0.37, 0, 0.44];
bf4 = AffineMap[ 0. Degree, 0. Degree, 0, 0.16, 0, 0];

fern = IFS[ {bf1, bf2, bf3, bf4}, Probabilities -> {0.73, 0.13, 0.11, 0.03} ]

chapter12 := chapter12 =
ChaosGame[fern, 50000, PlotStyle -> PointSize[0.0015]]

(* Random walk *)
```

```
appendixA := RandomWalk[5000]

(* Minimal Surface II *)

appendixB := appendixB =
ParametricPlot3D[
    {(r^2*Cos[2*phi])/2 - Log[r], -phi - (r^2*Sin[2*phi])/2, 2*r*Cos[phi]},
    {r, 0.0004, 2}, {phi, -2Pi, 3Pi}, PlotPoints -> {12, 100},
    ViewPoint->{-2.1, -1.1, 1.2} ]

(* Sierpinski sponge *)

(* Order: left, right, front, back, bottom, top *)

cheese[0, False, False, False, __ ] := {} (* small optimization *)

cheese[0, rt_, fr_, tp_, x0_, y0_, z0_] :=
  With[{xs = x0+1, ys = y0+1, zs = z0+1},
    { If[rt, Polygon[{{xs,y0,z0}, {xs,ys,z0}, {xs,ys,zs}, {xs,y0,zs}}], {}],
      If[fr, Polygon[{{x0,y0,z0}, {xs,y0,z0}, {xs,y0,zs}, {x0,y0,zs}}], {}],
      If[tp, Polygon[{{x0,y0,zs}, {xs,y0,zs}, {xs,ys,zs}, {x0,ys,zs}}], {}]
    }]

cheese[n_, rt_, fr_, tp_, x0_, y0_, z0_] :=
  With[{s = 3^(n-1), t = 2 3^(n-1), n1 = n-1},
   With[{xs = x0 + s, xt = x0 + t,
         ys = y0 + s, yt = y0 + t,
         zs = z0 + s, zt = z0 + t},
   { (* bottom layer *)
    cheese[n1, False, fr, False, x0, y0, z0],
    cheese[n1, False, fr, True, xs, y0, z0],
    cheese[n1, rt, fr, False, xt, y0, z0],
    cheese[n1, True, False, True, x0, ys, z0],
    cheese[n1, rt, False, True, xt, ys, z0],
 (* cheese[n1, False, False, False, x0, yt, z0], invisible *)
    cheese[n1, False, True, True, xs, yt, z0],
    cheese[n1, rt, False, False, xt, yt, z0],
     (* middle layer *)
    cheese[n1, True, fr, False, x0, y0, zs],
    cheese[n1, rt, fr, False, xt, y0, zs],
    cheese[n1, True, True, False, x0, yt, zs],
    cheese[n1, rt, True, False, xt, yt, zs],
     (* tp layer *)
    cheese[n1, False, fr, tp, x0, y0, zt],
    cheese[n1, False, fr, tp, xs, y0, zt],
    cheese[n1, rt, fr, tp, xt, y0, zt],
    cheese[n1, True, False, tp, x0, ys, zt],
    cheese[n1, rt, False, tp, xt, ys, zt],
    cheese[n1, False, False, tp, x0, yt, zt],
    cheese[n1, False, True, tp, xs, yt, zt],
    cheese[n1, rt, False, tp, xt, yt, zt]
  }
 ]]

Sierpinski[n_Integer] :=
    Block[{polylist},
        polylist = {EdgeForm[],
        cheese[n, True, True, True, 0, 0, 0]};
        polylist = Flatten[ polylist ];
```

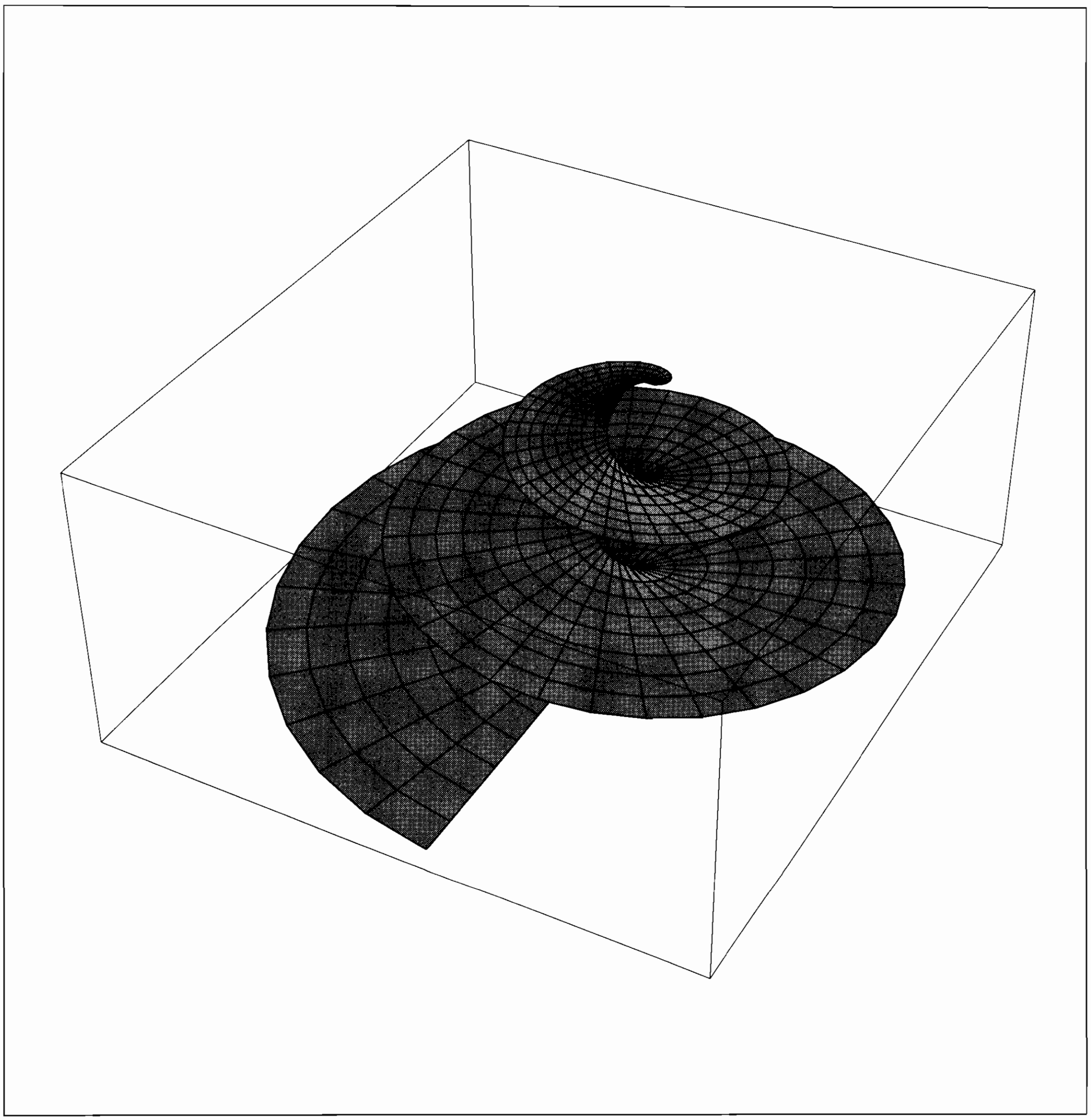
```
        Graphics3D[polylist, Boxed->False]
    ]   /; n >= 0

bookoptions :=
    Sequence[ViewPoint->{0.95, -3.1, 0.8},
        LightSources -> {{{1., 0., 5.}, RGBColor[1, 0, 0]},
                         {{1., 1., 1.}, RGBColor[0, 1, 0]},
                         {{0., 1., 0.4}, RGBColor[0, 0, 1]},
                         {{0., -1., 0.4}, RGBColor[0, 0, 1]}},
        Boxed->False, Background -> GrayLevel[0] ]

index := index =
Show[ Sierpinski[3], bookoptions ]

(* Cover *)

dia = 0.08;
exp = 0.4;
{u1, u2} = {0.001, 1};
{phi1, phi2} = 2 Pi{0, 2.43};
phi3 = phi2 + 2Pi/3;

res = 20;
thick = 0.0015

rs[u_, phi_] := dia u Exp[phi/(2Pi)]
hs[u_, phi_] := u (phi+1)^-exp

param[u_, phi_] :=
    { rs[u, phi] Cos[phi], rs[u, phi] Sin[phi], hs[u, phi] }

cover3 := cover3 =
    Module[{surf, lines, line, liner, phis, us, u, phi, max, maz, pr},
      surf = ParametricPlot3D[Evaluate[param[u, phi]],
               {u, u1, u2, 2(u2-u1)/res}, {phi, phi1, phi2, Pi/res},
               DisplayFunction -> Identity ];
      us = Range[-u2, -u1, 2(u2-u1)/res];
      lines = ParametricPlot3D[Evaluate[Function[u, param[u, phi]] /@ us],
               {phi, phi1, phi2, Pi/res},
               DisplayFunction -> Identity ];
      line = ParametricPlot3D[Evaluate[param[1.001u2, phi]],
               {phi, phi1, phi3, Pi/res},
               DisplayFunction -> Identity ];
      phis = Range[phi2, phi3, Pi/res];
      liner = ParametricPlot3D[Evaluate[Function[phi, param[u, phi]] /@ phis],
               {u, u1, 1.001u2, 2(1.001u2-u1)/res},
               DisplayFunction -> Identity ];
      max = rs[u2, phi3]; maz = hs[u2, phi1];
      pr = {-#,#}& /@ {max, max, maz};
      Show[ Graphics3D[{EdgeForm[], Thickness[thick],
                        surf[[1]], line[[1]], lines[[1]], liner[[1]]}],
            PlotRange -> pr, ViewPoint -> {1.7,-2,1.2},
            Boxed -> False ]
    ]

cover := cover = cover3 (* alias *)

(* cover of 2nd edition: exponentially shrinking torus *)
```

```
torus[ R_, r_, psi_, phi_, h_ ] :=
    { (R + r Cos[psi]) Cos[phi],
      (R + r Cos[psi]) Sin[phi],
      r Sin[psi],
      FaceForm[ ColorCircle[h], ColorCircle[h, 0.6] ]
    }

segment[ {phi0_, phi1_, dphi_}, {psi0_, psi1_, dpsi_} ] :=
    ParametricPlot3D[
        Evaluate[torus[1.2, Exp[-phi/(3Pi)], psi + phi/8, phi, psi-3Pi/4]],
        {phi, phi0, phi1, dphi}, {psi, psi0, psi1, dpsi},
        DisplayFunction -> Identity ]

cover2 := cover2 =
 Module[{glist, dphi = 2Pi/36, dpsi = 2Pi/32},
    glist = {segment[{-Pi/4,    0, dphi}, {Pi/2, 2Pi, dpsi}],
             segment[{  0,  3Pi/2, dphi}, {0, 2Pi, dpsi}],
             segment[{3Pi/2,  2Pi, dphi}, {1Pi/4, 7Pi/4, dpsi}],
             segment[{2Pi,  7Pi/2, dphi}, {0, 2Pi, 2dpsi}],
             segment[{7Pi/2,  4Pi, dphi}, {0, 6Pi/4, 2dpsi}],
             segment[{4Pi, 11Pi/2, dphi}, {0, 2Pi, 4dpsi}],
             segment[{11Pi/2, 6Pi, dphi}, {-Pi/4, 5Pi/4, 4dpsi}],
             segment[{6Pi, 17Pi/2, dphi/2}, {0, 2Pi, 4dpsi}]};
    Show[ glist, Boxed -> False, Lighting -> False, PlotRange -> All,
          DisplayFunction -> $DisplayFunction ] ]

(* cover of first edition: Maeder's shell *)

t0 = 0.001; t1 = N[ Pi - t0 ]
dt = (t1 - t0)/36; dp = N[ Pi/20 ]

part[t0_, phi0_, phi1_] := Block[{theta, phi},
    SphericalPlot3D[{Sin[theta] (2+Cos[phi/2]),
                     FaceForm[ColorCircle[phi/2, 1], ColorCircle[phi/2, 0.7]]},
                    {theta, t0, t1, dt}, {phi, phi0, phi1, dp},
                    DisplayFunction -> Identity] ]

cover1 := cover1 =
Module[{glist},
    glist = {part[t0, 0, 3Pi/2],     part[Pi/2, 3Pi/2, 4Pi/2],
             part[t0, 4Pi/2, 7Pi/2], part[Pi/2, 7Pi/2, 8Pi/2]};
   Show[ glist, Boxed -> False, Lighting -> False,
         DisplayFunction -> $DisplayFunction ] ]
```

Listing 10.3–1: BookPictures.m

Chapter 11

Notebooks



This chapter treats some of the issues concerning the use of *Mathematica* on computers with or without the notebook frontend. In Version 3, notebooks are *Mathematica* expressions themselves, which opens a whole range of new applications, because notebooks can be manipulated by programs.

Section 1 compares notebooks and packages and mentions a few things to keep in mind when working on different systems. It also suggests a way of developing packages as notebooks and discusses a template notebook and documentation issues.

Section 2 discusses the structure of a notebook as a *Mathematica* expression. Knowledge of this structure is necessary if you want to write programs that manipulate notebooks or if you want to use any of the more advanced features of the frontend.

In Section 3 we look at a few selected topics in frontend programming. We show how you can design buttons that perform an action when you click on them with the mouse. Another topic is the manipulation of whole notebooks within the kernel and an example of kernel–frontend interaction: automatic animation.

About the illustration overleaf:

A double spiral staircase.

```
ParametricPlot3D[{r (1 + phi/2) Cos[phi], r (1 + phi/2) Sin[phi], -phi/2},
  {r, 0.1, 1.1, 0.125}, {phi, 0, 11Pi/2, Pi/16}]
```

11.1 Packages and Notebooks

Notebooks are structured documents that can contain *Mathematica* input and output, graphics, and ordinary text. You can structure the information in the same way that it is structured in a book, defining chapters, sections, subsections, and so on. Notebooks are a feature of *Mathematica frontends*, programs that provide a sophisticated interface between the user and the *kernel*, which does the computations. Without the frontend, you work with *Mathematica* by typing input at the `In[`*n*`]:=` prompt and the results are displayed at the next `Out[`*n*`] =` prompt. This is the method of interaction used for the "live" calculations throughout this book.

11.1.1 Notebooks as Packages

A typical notebook consists of definitions and examples of their use. You probably did not bother to set up package contexts for the definitions in the smaller notebooks you wrote. If you want to use the definitions in your notebooks in other places as well, you should design them in the same way that you design packages. This section describes how to develop a package in the form of a notebook.

The package part of a notebook is everything in the initialization cells. These are the cells that have the option `InitializationCell->True`. In the implementation part you can use the same framework for setting up the contexts as you do in a package. The commands `BeginPackage[]`, `Begin[]`, `EndPackage[]`, and `End[]` should be put into separate cells. In the examples in this book, we have used blank lines to separate the parts of a package. These parts are best put into separate cells. Comments can go into text cells without the notation `(*` *comment* `*)`. If you decide to write a package in the form of a notebook you can make use of the advanced text-formatting and outlining capabilities of the frontend to annotate your code. You can also include examples of the use of your package and provide additional documentation. A template for such an annotated package is in the notebook Template.nb. Its structure is shown in Figure 11.1–1. The reference section is equivalent to the corresponding section in the plain-file package template Skeleton.m in Section 2.4.

The setup section contains commands needed to read in packages if the notebook contains examples (rather than a package). It can be deleted for a notebook that serves as a package. The interface, implementation, and epilog sections contain the package proper. Their contents are identical to the skeletal package Skeleton.m (see Listing 2.4–1), with added annotations and subsection headings.

We used this setup for one of our own packages. The notebook ChaosGame.nb contains the package commands in initialization cells. Additionally, it contains documentation and examples.

Prior to Version 3, a notebook could be read directly into the kernel, and all initialization cells were evaluated in the same way that they were when read from a plain file. Because

Package Template

by Roman E. Maeder

This notebook is a template for package and notebook development.

■ Reference

■ Setup

■ Interface

This part declares the publicly visible functions, options, and values.

■ Set up the package context, including public imports

■ Usage messages for the exported functions and the context itself

■ Error messages for the exported objects

■ Implementation

This part contains the actual definitions and any auxiliary functions that should not be visible outside.

■ Begin the private context (implementation part)

■ Read in any hidden imports

■ Unprotect any system functions for which definitions will be made

■ Definition of auxiliary functions and local (static) variables

■ Definition of the exported functions

■ Definitions for system functions

■ Restore protection of system symbols

■ End the private context

■ Epilog

This section protects exported symbols and ends the package.

■ Protect exported symbol

■ End the package context

■ Examples, Tests

Figure 11.1–1: Template.nb: An annotated package in a notebook

notebooks are now stored as *Mathematica* expressions themselves, this is no longer possible. Reading in a notebook does not evaluate its initialization cells, but reads in the whole notebook contents as an (inert) expression (see Section 11.2). The new autosave package mechanism is used to maintain the ability to develop packages as notebooks.

The package file ChaosGame.m that can be read into *Mathematica* using

```
Needs["ProgrammingInMathematica`ChaosGame`"]
```

is created automatically from the notebook ChaosGame.nb. It is updated every time the notebook is saved. The notebook option `AutoGeneratedPackage` causes this behavior. If it is set to `Automatic`, any cells that are marked as initialization cells (option `InitializationCell`) are written (in input form) to the package. With this setup, the notebook serves as the master copy of the package. Every time you modify the notebook, the package itself is updated automatically. If you create a new notebook from scratch and use any initialization cells you will be asked whether to generate the package the first time you save the notebook. The setting you specify is remembered as a notebook option. You can change it later with the option browser, if needed.

■ 11.1.2 Notebooks That Depend on Packages

You specify that a notebook needs a certain package in the same way that you do in other packages: put the command `<<`*Package*`.m` in your notebook, or better yet, use `Needs["`*Package*`` ` ``"]`. This command is best put into an initialization cell. It will then be executed whenever you open the notebook or before you perform the first evaluation in the notebook. Some computers have rather strange ways of referring to files on their hard disk. Using `Needs["`*Package*`` ` ``"]` avoids all those problems because *Mathematica* knows how to convert a context name into a file name.

Initialization cells are used for two purposes. First, they mark cells that are part of a package in notebook format. Second, they mark cells that should be evaluated when a notebook is opened. The reason for this double use is historical. Only in the first case do you want to set `AutoGeneratedPackage` to cause a package to be written automatically every time you save the notebook.

■ 11.1.3 Developing Packages as Notebooks

There are two ways to develop your packages: as notebooks or as plain files. You can keep the package you are working on in an editor, make changes, and then read it back into the *Mathematica* session with `<<`*package*`` ` ``. We gave some hints on how to set things up during the debugging phase in Section 2.3.1 on page 41.

With the notebook frontend, you might want to set things up differently. You can enter your package code directly into a notebook, as explained in Section 11.1.1. In this case,

too, you should insert the statement `Clear[`*syms*`...]` near the beginning of the notebook under development and make the cells containing the context-switching statements *inactive*. Instead of reading in the new version after making some corrections, you select all the cells in the notebook that are part of the package and evaluate them. You can keep test examples for your code in the same notebook and then evaluate them again to see if the changes have fixed the problems.

My recommendation is to separate the *implementation part* (the package) from the examples. The package should be written according to the guidelines in this book (Chapter 2). The notebook with the examples can then import this package with a cell containing the command `Needs["`*package*`` ` ``"]`.

In a notebook it is possible to reevaluate earlier input cells. The *kernel history*, the sequence of commands passed to the kernel, then no longer corresponds to the order of cells in the notebook. Using `%` to refer to earlier computations becomes confusing under these circumstances, because `%` refers to the result last produced by the kernel, not the result appearing in the cell above the current cell.

■ 11.1.4 Documentation Notebooks

Notebooks can also serve as on-line documentation. *Mathematica*'s help browser allows to view properly indexed notebooks and search for topics. Figure 11.1–2 shows the help browser displaying a help notebook from Chapter 12. When you develop a package you can design your own on-line help and have it show up in the help browser of the users of your package, provided the package is properly installed. Section 12.4 describes how help notebooks need to written and installed.

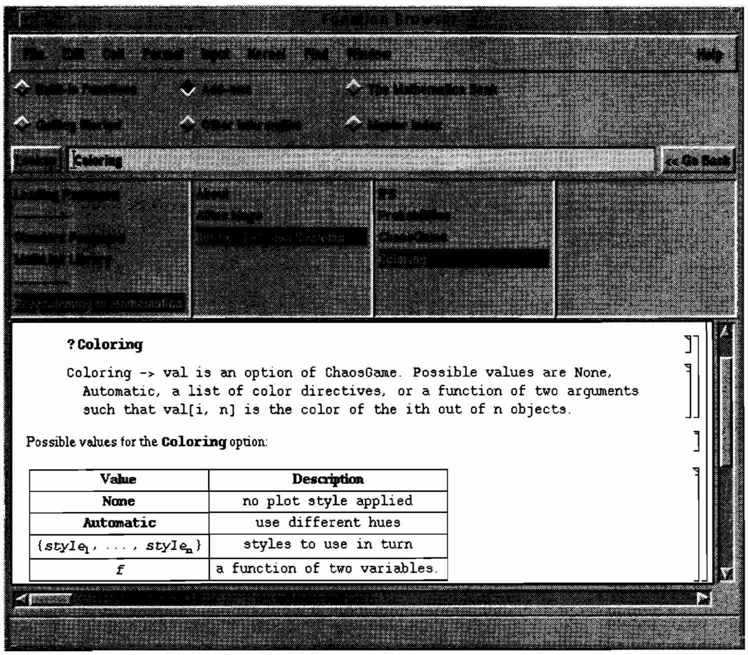


Figure 11.1–2: The help browser showing a topic from this book

■ 11.2 The Structure of Notebooks

A notebook is represented as a *Mathematica* expression. With hindsight, this representation is the most reasonable choice. In this form it is rather simple to write *Mathematica* programs that manipulate notebooks, because they are written in the same language. A notebook expression is a data structure. Just as a list simply contains its elements and does not perform any operations on them, a notebook is a container for the descriptions of the notebook's contents. A notebook file has the structure shown in Listing 11.2–1 (you can look at a notebook file with any text editor).

```
(***********************************************************
              Mathematica-Compatible Notebook
 ⋮
***********************************************************)

(*CacheID: number *)

Notebook[{
   cells and groups
    ⋮
 },
  notebook options ...
]

(*
  cache data
   ⋮
*)


(***********************************************************
              End of Mathematica Notebook file.
***********************************************************)
```

Listing 11.2–1: Structure of a notebook

If you read a notebook into the kernel (using << *file*`.nb`), everything but the `Notebook[]` expression is ignored by the kernel (because everything else is inside comments).

The initial comment identifies the file as a notebook in cases where it is manipulated outside *Mathematica* (for example, if it is sent by email). The cache ID comment tells the frontend that the cache data later in the file is valid. If you are bold enough to edit a notebook file with an ordinary text editor, you must delete this comment; otherwise, the frontend will rely on invalid cached data, which leads to unpredictable results (which is

computer scientists' preferred way to say that you will see garbage on the screen).

The notebook expression proper is a list of cells followed by a sequence of option settings for the notebook. These elements are described further in the remainder of this section. The cache data that follows helps the frontend to open and display the notebook faster. Because the frontend can infer the structure of the notebook from the data, it does not need to read the whole file to display a portion of it.

We can read the expression corresponding the the template notebook Template.nb (see Figure 11.1–1) into the kernel. No evaluation takes place; `Notebook[]` is only a container for its elements.

```
In[1]:= Short[ nb = << Template.nb, 6 ]
Out[1]//Short=
 Notebook[{Cell[CellGroupData[{Cell[Package Template,
       Title, ShowCellBracket -> False,
       TextAlignment -> Center],
      Cell[by Roman E. Maeder, Subsubtitle,
       ShowCellBracket -> False, TextAlignment -> Center]\
       , <<6>>, Cell[CellGroupData[<<2>>]]}, Open]]},
  <<11>>]
```

A notebook expression can be written to a file with `Put[`*expr*`, `*file*`.nb]`. The resulting file can be opened by the frontend. The missing cache data are restored automatically.

```
In[2]:= nb >> new.nb
```

Prior to Version 3, reading a notebook into the kernel caused all initialization cells to be evaluated. This functionality is no longer provided in this form. Instead, you can have the frontend write out a package containing all initialization cells. This package can then be read into the kernel. See Section 11.1.1.

Notebooks and cells are often rather large expressions. The formats defined in NotebookStuff.m (shown in Listing 11.2–2) cause notebooks, cells, and closed groups to be printed in an abbreviated form, similar to the way graphic expressions are printed (the familiar `-Graphics-` form).

```
skel = "[SkeletonIndicator]"

Format[Notebook[cells_List, ___]] := SequenceForm[skel, "Notebook", skel]

Format[Cell[c_, style_, ___]] := SequenceForm[skel, style, skel]

Format[Cell[CellGroupData[c_List, Closed]]] :=
    SequenceForm[skel, "closed group", skel]
```

Listing 11.2–2: Part of NotebookStuff.m: Abbreviated formats for notebook and cell expressions

With these formats, a notebook expression prints in a compact way.

```
In[3]:= nb
Out[3]= -Notebook-
```

The notebook contains a single (open) group of cells. Note that cells are identified by their style. Closed groups also are shown in abbreviated form.

```
In[4]:= nb[[1]]
Out[4]= {Cell[CellGroupData[{-Title-, -Subsubtitle-,
     -Text-, -closed group-, -closed group-,
     Cell[CellGroupData[{-Section-, -Text-,
        -closed group-, -closed group-, -closed group-},
       Open]], Cell[CellGroupData[{-Section-, -Text-,
        -closed group-, -closed group-, -closed group-,
        -closed group-, -closed group-, -closed group-,
        -closed group-, -closed group-}, Open]],
     Cell[CellGroupData[{-Section-, -Text-,
        -closed group-, -closed group-}, Open]],
     -closed group-}, Open]]}
```

■ 11.2.1 Cells

A cell has the structure `Cell[`*contents*`, "`*style*`", `*options*`...]`.

■ 11.2.1.1 Cell Contents

Cells contain text data, graphics, or box data. The data can have several forms. A complete list of these forms is in *The Mathematica Book*, Section 2.10.7.

`TextData[{`*text₁*`, ...}]`	a sequence of texts in various formats
`"text"`	plain text (short form of `TextData[{"text"}]`)
`BoxData[`*box*`]`	typesetting cell containing a single box (usual case)
`BoxData[{`*boxes*`...}]`	typesetting cell containing several boxes

Contents of cells

The individual $text_i$ in the list in `TextData[]` can be any of the following:

`"`*text*`"`	text in the current cell's style
`StyleBox["`*text*`", `*options*`...]`	text with particular option settings
`StyleBox["`*text*`", "`*style*`"]`	text rendered in a particular style
`Cell[`*contents*`, "`*style*`"]`	an inline cell in a particular style
`Cell[`*contents*`]`	an inline cell in the default style

Forms of text data

You can modify a notebook read into the kernel by giving rules that change some of the cells in the notebook. Here are a few patterns that are useful in the left side of such rules.

`Cell[c_, t:"`*style*`", opts___?OptionQ]`
: matches any cell in the given *style*. The tag `t` allows you to refer to this style on the right side of the rule without repeating its name string. This technique makes your code easier to maintain.

`Cell[c_String | TextData[{c___}], style_, opts___?OptionQ]`
: matches any cell containing text data. On the right side of the rule, you can use `{c}` to refer to the list of text data for either alternative. (The use of the pattern variable `c` as either a single expression or a sequence of expressions generates a harmless error message.)

`Cell[BoxData[{b___} | b_], style_, opts___?OptionQ]`
: matches any cell containing box data. On the right side of the rule, you can use `{b}` to refer to the list of boxes for either alternative.

`StyleBox[t_, opts1___?OptionQ, `*opt*`->val_, opts2___?OptionQ]`
: matches a stylebox in which the option *opt* is set. You can assemble a new stylebox in which *opt* has a new value with `StyleBox[t, opts1, `*opt*`->`*newval*`, opts2]` or remove *opt* with `StyleBox[t, opts1, opts2]`.

Examples of the use of such patterns are given in Section 11.3.2.

■ 11.2.1.2 Cell Groups

Cells can be grouped. The frontend indicates a group of cells with an additional cell bracket at the right margin that spans all cells in the group. A cell group is treated as contents of another dummy cell. The dummy cell enclosing the cell group data has no style specification, so a cell group is represented by

`Cell[CellGroupData[{`*cells*...`}, Open|Closed]]`.

If the cell group is closed, the frontend will display only the first cell in the group and represent the remaining cells by a special mark at the bottom of the cell group bracket.

This rule closes all groups in the notebook.

```
In[5]:= nb //. CellGroupData[cells_, Open] :>
                 CellGroupData[cells, Closed]

Out[5]= -Notebook-
```

Because the outermost group has been closed as well, not much remains visible in our short output format.

```
In[6]:= %[[1]]

Out[6]= {-closed group-}
```

11.2.1.3 Cell Options

All attributes of a cell are primarily determined by the cell's style. A style represents a collection of option settings that is used for all cells using that style. These inherited settings can be overridden for a cell by giving a sequence of options in the form

Cell[*contents*, "*style*", opt_1->val_1, opt_2->val_2, ...].

For example, the text cell

Cell[*text*, "Text", TextJustification->1]

will be displayed with nicely justified right margins, overriding the default of the "Text" style, which is ragged right margins.

The main tool for manipulating options is the frontend's option inspector. If you select a cell and set the inspector's scope to "selection" and its display mode to "as text," you can see which options are explicitly set for the selected cell.

11.2.2 Notebook Options

A notebook as a whole has options, too. They define those aspects of the notebook that differ from the global notebook frontend settings. Most often you set these options using one of the many menu commands of the frontend or with the option inspector.

Standard option manipulation tools can be used to obtain the options of a notebook read into the kernel. Note that the frontend adds a few undocumented options of its own to the ones you set yourself.

```
In[7]:= Short[ Options[ nb ], 5 ]
Out[7]//Short=
 {FrontEndVersion -> X 3.0 Beta 3,
  ScreenRectangle -> {{0, 1280}, {0, 1024}},
  AutoGeneratedPackage -> Automatic, <<6>>,
  ShowCellLabel -> True,
  RenderingOptions ->
   {ObjectDithering -> True, RasterDithering -> False}}
```

11.3 Frontend Programming

A thorough discussion of the frontend's programming interface is beyond the scope of this book. The following sections discuss a few selected topics and concentrate on the interaction of frontend and kernel. First we have a look at active elements (buttons) that allow you to design simple graphical user interfaces (GUIs) with *Mathematica*. Then, we discuss how you can develop programs running in the kernel that manipulate notebooks or the frontend. This capability makes it possible to generate notebooks automatically and to program the frontend for computer-assisted instruction (CAI), for example. Finally, we shall have a look at animation again, continuing the discussion started in Section 10.2.

11.3.1 Active Elements

The basic active element is the button box. A button box is displayed as a rectangular area. When you click on an active button box, the corresponding action is triggered.

11.3.1.1 Buttons

A button box has this structure:

`ButtonBox[`*contents*`,` *options*`...].`

The *contents* is the typeset expression that is displayed inside the button area. The following options can be given:

`ButtonFunction`	the action to perform when the button is activated (a pure function)
`ButtonSource`	the first argument of the button function
`ButtonData`	the second argument of the button function
`ButtonEvaluator`	where to evaluate the button function
`ButtonNote`	status line text when the mouse is over the button
`Active`	whether the button is active
`ButtonStyle`	style from which button options are inherited

Options of button boxes

When you click over an active button, the specified button function is called with two arguments, given by `ButtonSource` and `ButtonData`, respectively. The default button source is the button's contents. The default button data is `Null`.

With the setting `ButtonEvaluator->None` (the default), the button function is handled by the frontend itself. In this case it must have the form

`FrontEndExecute[{`*fecmds...*`}]&`

where *fecmds* are valid frontend commands; see Section 11.3.3.1.

With the setting `ButtonEvaluator->Automatic`, the button function is sent to the kernel for evaluation. This form is needed for complicated actions that cannot be handled by the frontend itself.

A button function is triggered (by clicking with the mouse inside the button) only if the button is *active*. A button is active if it has the option setting `Active->True`, or if it is inside an active cell.

Common button actions are predefined as styles. To make a button with such a predefined action, you need not give any of the options just described; you simply specify the style from which options should be inherited with `ButtonStyle->"`*style*`"`.

The default button style, `"Paste"`, for example, simply sets the button function to

```
ButtonFunction :> (
  FrontEndExecute[{
      FrontEnd`NotebookApply[FrontEnd`InputNotebook[], #, After]
    }]&)
```

The effect is to replace the selection in the current notebook by the contents of the button and to leave the insertion point after the inserted material. If the button contents contains a selection placeholder (shown as a black square), the current selection is inserted in its place, rather than discarded. This makes it easy to create buttons that wrap their contents around the selection. The Basic Input palette consists essentially of a large collection of such buttons.

The `"Evaluate"` button style defines this button function:

```
ButtonFunction :> (
  FrontEndExecute[{
      FrontEnd`NotebookApply[FrontEnd`InputNotebook[], #, All],
      SelectionEvaluate[FrontEnd`InputNotebook[], All]
    }]&)
```

The first action replaces the selection by the button contents (as before), but leaves the inserted material selected. The second action evaluates the selected material. Note that the button action is executed directly by the frontend, but the resulting evaluation will involve the kernel, of course.

In the next subsection we investigate another button style: hyperlinks.

■ 11.3.1.2 Example: Creating Hyperlinks

The "Hyperlink" button style causes a jump to a different part of the current notebook, or to another notebook (which is opened, if necessary). The button options that achieve this action are the following:

```
ButtonFunction :> (
  FrontEndExecute[{
     FrontEnd`NotebookLocate[{FrontEnd`ButtonNotebook[], #2}]
    }]&)
Active -> True
ButtonNote -> ButtonData
```

This button function makes use of the second argument supplied, `ButtonData`. Its value is either a string denoting a target address in the current notebook or a list {*file*, *target*} specifying a target in another notebook.

Single hyperlinks are best created using the frontend's Input ▷ Create Hyperlink menu command. Here, we want to use *Mathematica* to generate a whole matrix of similar hyperlinks; we want to have one hyperlink for each file in a directory. You can then simply click on one of the buttons to open the corresponding file. A hyperlink to a file *file* in the directory *dir* must look like this:

`ButtonBox[`*label*`, ButtonData -> {"`*dir/file*`", None}, ButtonStyle->"Hyperlink"]`,

where the label is constructed from the file name and has the form

`StyleBox["\"`*file*`\"", ShowStringCharacters -> False]`.

(The file name should not be typeset as an expression, but as a string. Strings require extra quotes in typeset expressions, but we do not want to see them; therefore, we suppress their rendering with the style box.) The special target `None` is used because we do not want to jump to a particular place in the notebook.

Let us create a matrix of such buttons from a list of file names. First we develop an auxiliary function `makeHyperlink[`*file*`]` that creates a single button. It is shown in Listing 11.3–1.

Let us now create a notebook with a list of hyperlinks to all our book packages.

We make the directory with the packages for *Programming in Mathematica* our current directory. `ToFileName[{`dir_1`, ...}]` assembles a directory hierarchy in a machine-independent way. `$TopDirectory` contains the directory where *Mathematica* is installed.

```
In[1]:= SetDirectory[ ToFileName[{$TopDirectory, "AddOns",
             "ExtraPackages", "ProgrammingInMathematica"}]
        ];
```

We obtain a list of all packages in this directory.

```
In[2]:= (packages = FileNames["*.m"]) // Short
Out[2]//Short=
 {Abs.m, AffineMaps.m, AlgExp.m, Atoms.m,
  AutoAnimation.m, <<55>>, VectorCalculus.m, WrapHold.m}
```

```
makeHyperlink::usage = "makeHyperlink[filename, tag:None] gives a button box
    that acts as a hyperlink to filename."

Options[makeHyperlink] = {
    Directory -> "",
    ButtonStyle -> "Hyperlink"
}

buttonOpts = {ButtonStyle, ButtonData, ButtonNote, ButtonFunction, ButtonEvaluator}
q = "\""

makeHyperlink[name_String, tag_:None, opts___?OptionQ] :=
    With[{ dir = Directory /. {opts} /. Options[makeHyperlink],
           label = StyleBox[q <> name <> q, ShowStringCharacters->False]
         },
        ButtonBox[ label, ButtonData->{dir <> name, tag},
            FilterOptions[buttonOpts, opts, Options[makeHyperlink]],
            ButtonNote->name ]
    ]
```

Listing 11.3–1: Part of NotebookStuff.m: Making buttons

We return to the previous working directory.

```
In[3]:= ResetDirectory[];
```

We turn the file names into hyperlink buttons.

```
In[4]:= buttons = makeHyperlink /@ packages;
```

We want to assemble them in a matrix with three columns.

```
In[5]:= cols = 4;
```

We append the necessary number of dummy buttons to make the number of buttons divisible by `cols`.

```
In[6]:= buttons =
          Join[buttons, Table[ButtonBox[""],
            {Mod[-Length[buttons], cols]}]];
```

Here is the matrix. Transposition makes the entries run down the columns.

```
In[7]:= matrix = Transpose[
            Partition[buttons, Length[buttons]/cols]
        ];
```

We put the button matrix into a grid box.

```
In[8]:= gridbox = GridBox[ matrix, GridFrame->True ];
```

We put the grid box into a small notebook consisting of just one cell. The output file packages.nb can be opened with the frontend and the grid box can be pasted into another notebook.

```
In[9]:= Notebook[{Cell[BoxData[gridbox], "Text"]}] >>
            packages.nb
```

Figure 11.3–1 shows the resulting notebook packages.nb. If you click on one of the buttons, the named package will be opened.

■ 11.3.2 Manipulating a Notebook in the Kernel

In Section 11.2.1.2, we saw an example of a modification of a notebook in the kernel. The frontend makes it easy to change option settings for single cells or selections of cells. More systematic modifications are often easier to perform, however, with a few rule applications in the kernel. As shown, you can read in a notebook into the kernel, apply transformation

Abs.m	ComplexMap2.m	NotebookLog.m	ReIm.m
AffineMaps.m	ComplexMap3.m	NotebookStuff.m	RungeKutta.m
AlgExp.m	ComplexMap4.m	OddEvenRules.m	SessionLog.m
Atoms.m	ComplexMap5.m	Options.m	Skeleton.m
AutoAnimation.m	ComplexMap.m	OptionUse.m	SphericalCurve.m
BadExample.m	ContinuedFraction.m	Package1.m	Struve.m
BestExample.m	ExpandBoth.m	Package2.m	SwinnertonDyer.m
BetterExample.m	Fibonacci1.m	Package3.m	Tensors.m
BinarySearch1.m	FoldRight.m	ParametricPlot3D.m	TrigDefine.m
BinarySearch2.m	GetNumber.m	Plot.m	TrigFormats.m
BookPictures.m	IFS.m	PrimePi.m	TrigSimplification.m
CartesianMap1.m	init.m	PrintTime.m	Until.m
CartesianMap2.m	MakeFunctions.m	RandomWalk.m	VectorCalculus.m
ChaosGame.m	MakeMaster.m	ReadList.m	WrapHold.m
Collatz.m	Newton1.m	ReadLoop1.m	
ComplexMap1.m	Newton.m	ReadLoop2.m	

Figure 11.3–1: packages.nb: A matrix of hyperlink buttons

rules to it, and write it back out to a file. You can then open this new notebook in the frontend. Section 11.2.1.1 listed a few useful patterns that match cells with certain properties. Here, then, are the corresponding rules that transform the matching cells in a certain way.

In this list of sample rules *nb* denotes a notebook object, typically obtained from reading in a notebook file.

nb `/. Cell[c_, "Section", opts___] :> Cell[c, "Subsection", opts]`
turns all sections into subsections by replacing the style of the matching cells.

`DeleteCases[`*nb*`, Cell[_, "Message", ___], -2]`
deletes all (error) message cells. The level specification `-2` restricts the search for matching cells to the innermost cells in groups, which speeds up the operation.

nb `//. CellGroupData[c_, Open] :> CellGroupData[c, Closed]`
closes all groups (note the use of `//.` because groups can be nested).

nb `/. Cell[c_, t:"Text", opts___] :> Cell[c, t, TextJustification->1, opts]`
justifies text in all text cells.

nb `/. StyleBox[t_, opts1___?OptionQ, FontSlant->"Italic", opts2___] :>`
`StyleBox[t, opts1, FontWeight->"Bold", opts2]`
set all text in a slanted font in bold face instead.

For these kinds of tasks, rule-based programming is the only reasonable programming style. You would not want to wade through a notebook expression in some complicated loop with many conditional statements.

You can also create a new notebook within the kernel. We saw an example, a session transcript, in Section 9.4.3.

■ 11.3.3 Kernel Interaction

The frontend and kernel communicate via *MathLink*. In a kernel controlled by a frontend, the global variable `$FrontEnd` gives the frontend object representing the controlling frontend. One of the ingredients of a frontend object is the name of the link between frontend and kernel. The kernel can program the frontend by sending expressions that are valid frontend commands through this link.

The various notebooks that the frontend maintains are represented by notebook objects. They contain a reference to the frontend object controlling them, and therefore you can also send commands to notebooks.

■ 11.3.3.1 Frontend Commands

The frontend has its own set of commands. Every action triggered, by a menu choice, for example, can be expressed in the frontend's own programming language. Most of these actions have a corresponding command in the kernel. The two commands have the same name, but they live in different contexts. The kernel version lives in the usual `System`` context; the frontend version lives in the `FrontEnd`` context. The kernel command is usually a simple wrapper that sends the corresponding frontend command to the frontend, using `$FrontEnd`, which stands for the frontend connected to the kernel, or directly to a notebook object, such as `EvaluationNotebook[]`, which stands for the notebook from which the current kernel evaluation was started.

The command `SelectionMove[`*notebookobject*`,` *direction*`,` *unit*`,` *count*`]`, for example, which causes the current selection in the given notebook object to be moved as indicated by the additional arguments, is implemented in the kernel essentially as

```
SelectionMove[note_NotebookObject, dir_, unit_, cnt_:1] :=
    write[ note, FrontEnd`SelectionMove[note, dir, unit, cnt]]
```

(the auxiliary function `write[]` extracts the *MathLink* connection inside the notebook object and writes the command to the link.) The frontend commands in the `FrontEnd`` context have no effect in the kernel, and the kernel versions have no effect in the frontend. In Section 11.3.1.1 you saw how to invoke frontend commands directly inside the frontend, without help from a kernel. The frontend has no built-in evaluator, however, so any more complicated action (such as figuring out `Plus[1,1]`) must be done in the kernel.

The most important kernel commands for frontend manipulation are documented in Subsection 2.10.3 of the *Mathematica* book. The large number of special frontend commands appearing in the menus are largely undocumented and should be used with care. You can use `FrontEndExecute[`*command*`]` to send a command to the frontend.

■ 11.3.3.2 Example: A Button That Creates Hyperlinks

In Section 11.3.1.2 we saw how to create a grid of hyperlink buttons. Now we want to write a button that performs these steps and inserts the resulting grid into the current notebook.

The button that triggers the action simply calls a kernel function. It looks like this:

```
ButtonBox[
  StyleBox["\"Paste Directory Listing\"", ShowStringCharacters->False],
  ButtonFunction -> (directoryList&),
  ButtonEvaluator -> Automatic,
  Active -> True
]
```

The button function does not need its argument; therefore, it is a constant pure function. The option `ButtonEvaluator -> Automatic` causes the button function to be sent to the kernel for evaluation. The kernel, therefore, receives the command `directoryList`. The action is triggered with a delayed value on this symbol, as described in Section 5.5. The symbol `directoryList` needs to be defined in an initialization cell, so that it is set up when the notebook is opened. It looks like this:

```
Needs["ProgrammingInMathematica`NotebookStuff`"]

directory = "."
cols = 3

directoryList :=
    Module[{filenames, dir, buttons, matrix, box},
        SetDirectory[directory];
        dir = Directory[]; (* absolute path *)
        filenames = FileNames[{"*.m", "*.nb"}];
        ResetDirectory[];
        buttons = makeHyperlink[#, Directory->dir]& /@ filenames;
        buttons =  Join[buttons, Table[ButtonBox[""],
                     {Mod[-Length[buttons], cols]}]];
        matrix = Transpose[ Partition[buttons, Length[buttons]/cols] ];
        box = GridBox[matrix, GridFrame->True];
        NotebookWrite[EvaluationNotebook[], BoxForm[box]];
    ]
```

The code performs the same steps that we performed interactively in Section 11.3.1.2. First we obtain the names of all packages and notebooks in the desired directory (this is a bit tricky: directories in hyperlinks are relative to the directory of the notebook containing the hyperlink, rather than relative to the current directory of the kernel). Then, we create the matrix of buttons, put it into a grid box, and insert the grid box into the current notebook, at the insertion point.

The elements just described are in the notebook NotebookDemo.nb. You can open it in the frontend and play with the functions.

■ 11.3.4 Application: Automatic Animation

In Section 10.2 we saw how the animation commands, such as `Animate[]`, use two auxiliary functions, `$RasterFunction` and `$AnimationFunction`, to render and animate the frames constituting the animation. Their default values of `$RasterFunction = $DisplayFunction` and `$AnimationFunction = Identity` simply render all frames individually. The frontend puts a sequence of graphic cells into a group. To see the animation, you select the group and choose the Cell ▷ Animate Selected Graphics menu command. In this section we look at a program that can be assigned to `$AnimationFunction` to perform these steps automatically.

The commands in `$AnimationFunction` in AutoAnimation.m (Listing 11.3–2) perform the following steps:

- Obtain the notebook object corresponding to the notebook in which the current evaluation (the animation) occurred.
- Select all generated cells, that is, the graphics that constitute the animation. A generated cell (cell property `GeneratedCell`) is a cell that was sent to the frontend as a result or side effect of an evaluation.
- Extend the selection to comprise the group around the graphics cells.
- Close this group of graphics cells, so that only the first graphic remains visible. `FrontEndExecute[FrontEndToken["`*command*`"]]` issues a frontend command that is associated with a simple frontend action. `OpenCloseGroup` *toggles* the state of a group. Because the group was open when the graphics were generated, it is now closed.
- Issue the `SelectionAnimate` frontend command. A command is sent to the frontend with `FrontEndExecute[FrontEnd`*command*`]`. The variable `$AnimationTime` determines for how long the animation will run.
- Move the selection after the graphics cells.
- Return the list of graphics as result of the animation. This list is passed as the argument of `$AnimationFunction`; it is not needed otherwise in this application.

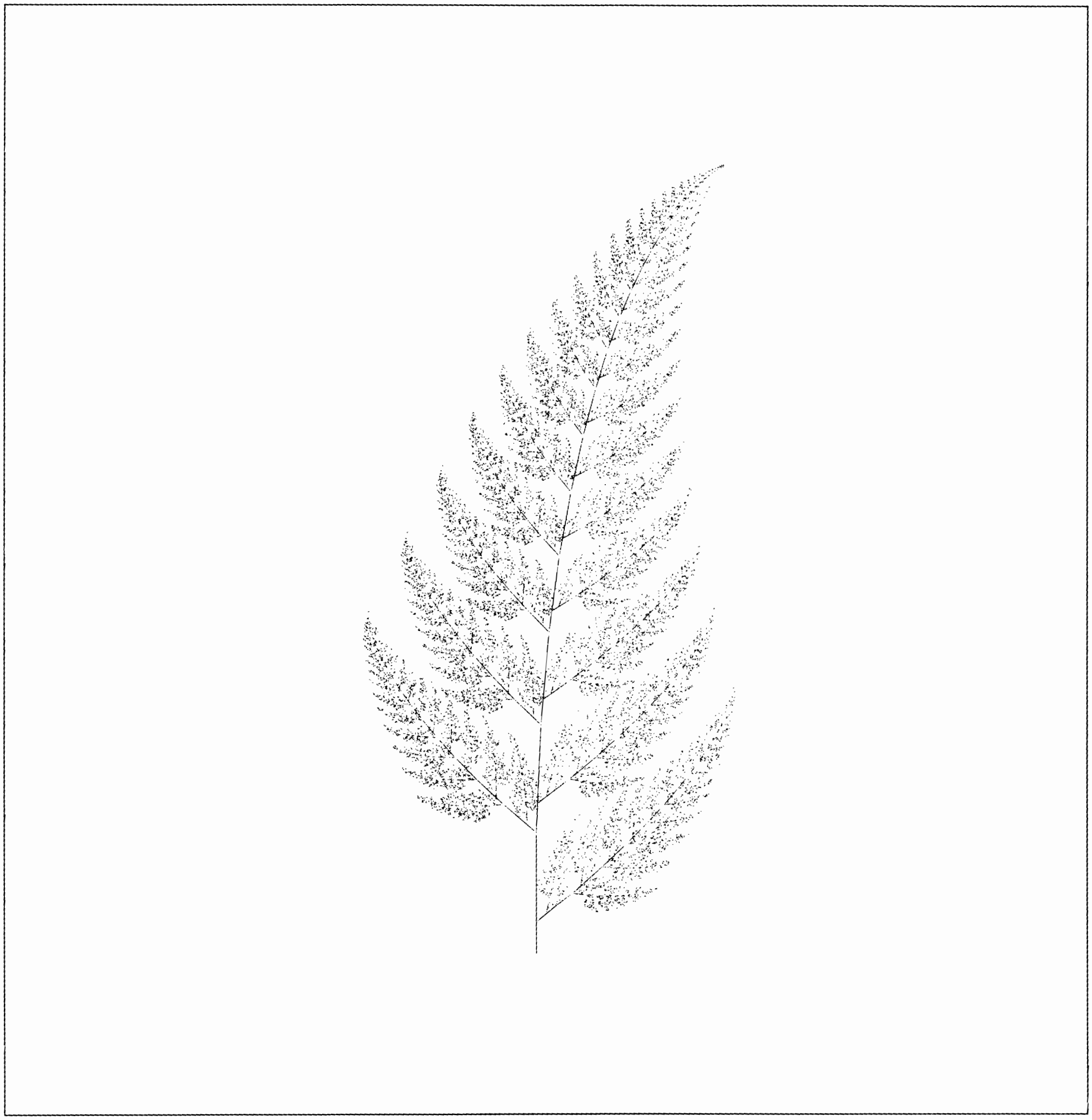
The frontend tokens, such as `OpenCloseGroup`, are not documented; you can consult the menu setup file SystemFiles/FrontEnd/TextResources/*machine*/MenuSetup.tr to find out which frontend commands correspond to menu entries.

```
System`$RasterFunction = $DisplayFunction

System`$AnimationTime = 60

System`$AnimationFunction =
    With[{nb = EvaluationNotebook[]},
        SelectionMove[nb, All, GeneratedCell];
        SelectionMove[nb, All, CellGroup];
        FrontEndExecute[FrontEndToken["OpenCloseGroup"]];
        FrontEndExecute[FrontEnd`SelectionAnimate[nb, $AnimationTime]];
        SelectionMove[nb, After, CellGroup];
        #
    ]&

Null
```

Listing 11.3–2: AutoAnimation.m: Automatic animation in the frontend

Chapter 12

Application: Iterated Function Systems



This chapter does not follow our usual principle of developing the code step by step; rather, we present a final, polished package that implements some basic functions for iterated function systems, one of the exciting topics of chaos and fractals.

Section 1 discusses affine maps, the basic tool underlying iterated function systems. We shall develop data types to represent affine maps and write code to apply such maps to coordinates and graphic objects.

Section 2 introduces iterated function systems and shows some of the phenomena that appear. The main topic is the invariant set and the chaos game, an efficient method to draw a picture of the invariant set.

In Section 3 we shall present examples of interesting iterated function systems: self-similar fractals and images of natural objects.

The last section discusses on-line documentation of application programs, such as our iterated–function-system package. The frontend provides tools for presenting on-line documentation of *Mathematica*'s built-in objects and of your own packages.

About the illustration overleaf:

One of the icons of fractals, Barnsley's fern.

```
bf1 = AffineMap[ -2.5 Degree, -2.5 Degree, 0.85, 0.85, 0, 1.6];
bf2 = AffineMap[ 49. Degree, 49. Degree, 0.3, 0.34, 0, 1.6];
bf3 = AffineMap[ 120. Degree, -50. Degree, 0.3, 0.37, 0, 0.44];
bf4 = AffineMap[ 0. Degree, 0. Degree, 0, 0.16, 0, 0];
fern = IFS[{bf1, bf2, bf3, bf4}, Probabilities -> {0.73, 0.13, 0.11, 0.03}]
ChaosGame[fern, 50000, PlotStyle -> PointSize[0.0015]];
```

12.1 Affine Maps

An affine map is a linear map in the plane. It can be described by a matrix

$$m = \begin{pmatrix} r\cos\varphi & -s\sin\psi & e \\ r\sin\varphi & s\cos\psi & f \end{pmatrix}. \tag{12.1–1}$$

The image of a point $\{x, y\}$ is computed by the dot product:

$$\{x', y'\} = m.\{x, y, 1\}, \tag{12.1–2}$$

that is,

$$\begin{aligned} x' &= xr\cos\varphi - ys\sin\psi + e \\ y' &= xr\sin\varphi + ys\cos\psi + f. \end{aligned} \tag{12.1–3}$$

Many sources quote the matrix 12.1–1 not in terms of r, s, φ, and ψ, but in terms of the resulting quantities as

$$m = \begin{pmatrix} a & b & e \\ c & d & f \end{pmatrix}. \tag{12.1–4}$$

12.1.1 A Data Type for Affine Maps

We can represent a linear map as a datum of the form `map[`*matrix*`]`, where *matrix* is the matrix from Equation 12.1–1. The constructor `AffineMap[`φ`, `ψ`, `r`, `s`, `e`, `f`]` computes the matrix and returns an affine map. A simple rule makes such maps behave like ordinary functions that can be applied to points. The rule for *map*`[{`x`, `y`}]` computes the image of the point $\{x, y\}$ according to Equation 12.1–2.

```
AffineMap[phi_, psi_, r_, s_, e_, f_] :=
    map[{{r Cos[phi], -s Sin[psi], e},
         {r Sin[phi],  s Cos[psi], f}}]
map[mat_?MatrixQ][{x_, y_}] := mat . {x, y, 1}
```

Several special choices of φ, ψ, r, s, e, and f have an intuitive geometric interpretation: A rotation by an angle α has $\varphi = \psi = \alpha$, $r = s = 1$, and $e = f = 0$; a scaling by a factor t has $\varphi = \psi = 0$, $r = s = t$, and $e = f = 0$; a translation by a vector $\{u, v\}$ has $\varphi = \psi = 0$, $r = s = 0$, $e = u$, and $f = v$.

In a functional language, such as *Mathematica*, there is an alternative way to define affine maps: you can simply specify a function of two arguments `x` and `y` in the form

`Function[{x, y}, {`$expr_x$`, `$expr_y$`}]`.

The two expressions $expr_x$ and $expr_y$ define the images of the `x` and `y` coordinates, respectively. A rotation by $\pi/2$, for example, can be written as `Function[{x, y}, {-y, x}]`.

To make use of this possibility, we allow a second form of the constructor: `AffineMap[{`x`, `y`}, {`$expr_x$`, `$expr_y$`}]`. Internally, we store the affine map as a pure function. The rule for applying it to a point is straightforward. Note that you are responsible for ensuring that the body `{`$expr_x$`, `$expr_y$`}` is in fact an affine (that is, linear) map. There is also a version of the constructor that takes a 2×3 matrix as argument. This matrix is used unchanged as the affine map. The package AffineMaps.m contains all these constructors; see Table 12.1–1.

`AffineMap[`φ`, `ψ`, `r`, `s`, `e`, `f`]`	affine map $\begin{pmatrix} r\cos\varphi & -s\sin\psi & e \\ r\sin\varphi & s\cos\psi & f \end{pmatrix}$
`AffineMap[{{`a`, `b`, `e`}, {`c`, `d`, `f`}}]`	affine map $\begin{pmatrix} a & b & e \\ c & d & f \end{pmatrix}$
`AffineMap[{`x`, `y`}, {`$expr_x$`, `$expr_y$`}]`	affine map given by a pure function
`rotation[`α`]`	rotation by α
`scale[`t`]`	scaling by a factor t
`scale[`r`, `s`]`	scaling by factors r and s, respectively
`translation[{`u`, `v`}]`	translation by $\{u, v\}$

Table 12.1–1: Generators for affine maps

The composition of two affine maps is again an affine map. No special code is required. *Mathematica* turns any expression `Composition[`m_1`, `m_2`][{`x`, `y`}]` into m_1`[`m_2`[{`x`, `y`}]]`. Our ordinary rules for applying maps to points can then take effect. Nevertheless, we provided some code to simplify composition.

This symbolic example shows the built-in rules for composition.

```
In[1]:= Composition[f, g][x]
Out[1]= f[g[x]]
```

The composition of two maps given by their matrices is found by matrix multiplication. This simplification will save us some time, if the same composition is applied many times to points.

```
In[2]:= Composition[ scale[0.5], translation[{1, 1}] ]
Out[2]= -map-
```

Maps are an abstract data type; their internals are hidden. To see what a map does, we can apply it to symbolic arguments.

```
In[3]:= %[{x, y}]
Out[3]= {0.5 + 0.5 x + 0. y, 0.5 + 0. x + 0.5 y}
```

■ 12.1.2 Transforming Geometric Objects

Our goal is to transform geometric objects, such as points, lines, polygons, and circles. Because we know how to transform points, we reduce the transformation of all other objects to the transformation of points.

To transform a `Point[{x, y}]` graphic object, we simply transform its coordinates and return another point object. The affine image of a line is another line, whose vertices are the transformed vertices of the original line. Therefore, we can simply use `Map (/@)` to apply the affine map to the points that describe the line. The same is true for polygons. Their image is another polygon (because the affine image of a straight line is another straight line).

```
(m_map)[Point[xy_]] := Point[m[xy]]

(m_map)[Line[points_]] := Line[m /@ points]

(m_map)[Polygon[points_]] := Polygon[m /@ points]
```

The affine image of a circle is an ellipse. The major axes of this ellipse are in general not parallel to the x and y axes; therefore we cannot use *Mathematica*'s `Ellipse[]` graphic primitive to express affine images of circles or ellipses (the major axes of `Ellipse[]` objects are always horizontal and vertical, respectively). The best we can do is approximate the circle (or ellipse) by a line with many points and then transform this polygon. The global variable `$CirclePoints` specifies how many points are to be used for these polygons. Disks can similarly be transformed into filled polygons. Please consult Listing 12.1–1 for the details.

Graphic directives that specify the size of graphic objects also can be transformed. The area contraction factor c of a linear map is the determinant of its matrix (disregarding the translation components):

$$c = \begin{vmatrix} a & b \\ c & d \end{vmatrix} . \tag{12.1–5}$$

If the map is given as a pure function, we need to compute the matrix elements first. The matrix is the Jacobian of the function, found by differentiation (see Section 4.7.3). Linear dimensions are contracted by a factor $\sqrt{c}$ on average (the exact value depends on the direction). We can use this factor to scale the arguments of the directives `PointSize`, `AbsolutePointSize`, `Thickness`, and `AbsoluteThickness`. The code is rather short:

```
AverageContraction[map[mat_?MatrixQ]] := Abs[Det[ Drop[#, -1]& /@ mat ]]
AverageContraction[map[f_Function]] :=
    Module[{x, y}, Abs[Det[ Outer[D, f[x, y], {x, y}] ]] ]

(m:map[_])[(h:PointSize|AbsolutePointSize|Thickness|AbsoluteThickness)[r_]] :=
    h[r Sqrt[AverageContraction[m]]]
```

Any other directives (specifying colors, etc.) are simply left unchanged. Complete `Graphics[{`*objs*`}, {`*opts*`}]` are transformed by applying the transformation to all elements of the graphics list `{`*objs*`}` with the definition

```
(m_map)[Graphics[objs_List, opts___]] :=
    Graphics[Function[g, m[g], Listable] /@ objs, opts]
```

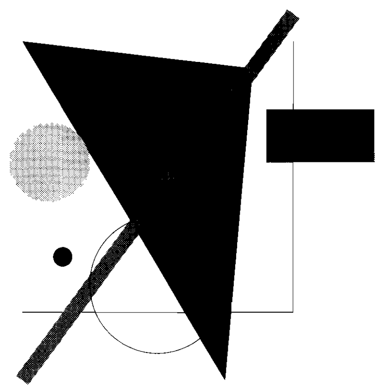
The construct `Function[g, m[g], Listable]`, instead of simply `m`, turns the affine map `m` temporarily into a listable function. It is needed because graphics lists can be nested (see Section 5.2.4).

Throughout this chapter, we shall work with these modified option settings for `Graphics`.

```
In[4]:= SetOptions[ Graphics, AspectRatio -> Automatic,
                              PlotRange -> All ];
```

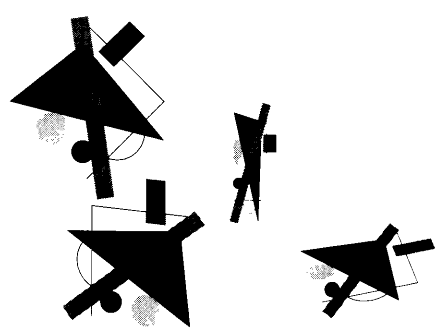
This graphic contains one of each kind of objects and directives for which we defined mappings. It serves as our test example.

```
In[5]:= gr1 = Show[ Graphics[{
         Line[{{-1, -1}, {1, -1}, {1, 1}}],
         Polygon[{{-1, 1}, {0.5, -1.5}, {0.7, 0.8}}],
         {PointSize[0.05], Point[{-0.7, -0.6}]},
         {Thickness[0.04], GrayLevel[0.6],
          Line[{{-1, -1.5}, {1, 1.2}}]},
         {GrayLevel[0.8], Disk[{-0.8,0.1}, 0.3]},
         {GrayLevel[0.4], Rectangle[{.8, .1}, {1.6, .5}]},
         Circle[{0, -0.8}, 0.5]
         }] ];
```



This image shows the effect of one of each kind of affine map. The translations help to separate the images of the different maps.

```
In[6]:= Show[
        Through[ {
          Composition[translation[{-3, 1}], rotation[Pi/4]],
          scale[0.3, 0.8],
          AffineMap[0.2, 0.4, 0.9, 0.5, 2, -2],
          AffineMap[{x, y}, {0.9 y - 2, x - 0.1 y - 2}]
         }[gr1] ] ];
```



Through[] applies functions in a list to a given argument and returns the list of results.

```
In[7]:= Through[ {f, g, h}[x] ]
Out[7]= {f[x], g[x], h[x]}
```

```
BeginPackage["ProgrammingInMathematica`AffineMaps`"]

AffineMap::usage = "AffineMap[\[Phi], \[Psi], r, s, e, f] generates an affine map
    with rotation angles \[Phi], \[Psi], scale factors r, s, and translation
    components e, f. AffineMap[{x, y}, {fxy, gxy}] generates an affine map
    with the two components given as expressions in x and y.
    AffineMap[matrix] uses the 2x3 matrix for the affine map."

map::usage = "-map- represents an affine map."

rotation::usage = "rotation[\[Alpha]] generates a rotation by \[Alpha]."
scale::usage = "scale[s, t] generates a scaling map with factors s and t.
    scale[r] scales both coordinates by r."
translation::usage = "translation[{x, y}] generates a translation by
    the vector {x, y}."

AverageContraction::usage = "AverageContraction[map] gives the average
    area contraction factor (the determinant) of an affine map."

$CirclePoints::usage = "$CirclePoints is the number of vertices of the
    polygon approximating the affine image of a circle."

$CirclePoints = 24

Begin["`Private`"]

(* affine map datatype *)

Format[m_map] := "-map-"

(* Terminology of Peitgen/Jürgens/Saupe *)

AffineMap[phi_, psi_, r_, s_, e_, f_] :=
    map[{{r Cos[phi], -s Sin[psi], e}, {r Sin[phi],  s Cos[psi], f}}]

(* as expressions. Does not test for affinity *)

AffineMap[params:{_Symbol, _Symbol}, expr:{_, _}] := map[ Function[params, expr] ]

(* matrix directly *)

AffineMap[ mat_?MatrixQ ] /; Dimensions[mat] == {2, 3} := map[ mat ]

(* apply to points *)

map[mat_?MatrixQ][{x_, y_}] := mat . {x, y, 1}
map[f_Function][{x_, y_}] := f[x, y]

(* simplify composition *)

map/: Composition[map[mat1_?MatrixQ], map[mat2_?MatrixQ]] :=
    map[ mat1 . Append[mat2, {0,0,1}] ]
map/: Composition[map[f_Function], map[g_Function]] :=
    Module[{x, y}, AffineMap[{x, y}, f @@ g[x, y]] ]

(* properties *)

AverageContraction[map[mat_?MatrixQ]] := Abs[Det[ Drop[#, -1]& /@ mat ]]
AverageContraction[map[f_Function]] :=
    Module[{x, y}, Abs[Det[ Outer[D, f[x, y], {x, y}] ]] ]

(* Graphic objects *)
```

```
(m_map)[Point[xy_]] := Point[m[xy]]

(m_map)[Line[points_]] := Line[m /@ points]

(m_map)[Polygon[points_]] := Polygon[m /@ points]

(* rectangles: convert to polygon *)

(m_map)[Rectangle[{xmin_, ymin_}, {xmax_, ymax_}]] :=
    m[Polygon[{{xmin, ymin}, {xmax, ymin}, {xmax, ymax}, {xmin, ymax}}]]

(* Circles/Ellipses: convert to lines/polygons *)

(m_map)[Circle[xy_, {rx_, ry_}]] :=
  With[{dp = N[2Pi/$CirclePoints]},
    m[ Line[ Table[xy + {rx Cos[phi], ry Sin[phi]},
                {phi, 0, 2Pi, dp}] ]
  ] ]

(m_map)[ Circle[xy_, r_] ] := m[ Circle[xy, {r, r}] ]

(m_map)[Disk[xy_, {rx_, ry_}]] :=
  With[{dp = N[2Pi/$CirclePoints]},
    m[ Polygon[ Table[xy + {rx Cos[phi], ry Sin[phi]},
                {phi, 0, 2Pi-dp, dp}] ]
  ] ]

(m_map)[ Disk[xy_, r_] ] := m[ Disk[xy, {r, r}] ]

(m_map)[ (Circle|Disk)[xy_, r_, args__] ] := Sequence[] (* not implemented *)

(* text: transform location *)

(m_map)[Text[text_, pos:{_, _}, args___]] := Text[text, m[pos], args]

(* not implemented: circular arcs, Raster, RasterArray,
   scaled coordinates, scaling of text *)

(* directives *)

(m_map)[(h:PointSize|AbsolutePointSize|Thickness|AbsoluteThickness)[r_]] :=
    h[r Sqrt[AverageContraction[m]]]

(* Graphics *)

(m_map)[Graphics[objs_List, opts___]] :=
    Graphics[Function[g, m[g], Listable] /@ objs, opts]

(* catchall *)

(m_map)[unknown_] := unknown

(* generators *)

rotation[alpha_] := AffineMap[alpha, alpha, 1, 1, 0, 0]

scale[s_, t_] := AffineMap[0, 0, s, t, 0, 0]
scale[r_] := scale[r, r]

translation[{x_, y_}] := AffineMap[0, 0, 1, 1, x, y]

End[ ]

Protect[ AffineMap, rotation, scale, translation, AverageContraction ]

EndPackage[ ]
```

Listing 12.1–1: AffineMaps.m: Affine maps of graphic objects

12.2 Iterated Function Systems

An *iterated function system* (IFS) is a collection of affine maps. An IFS operates on a set of points by transforming the set according to all maps in the system and then taking the union of the results. For an IFS $F = \{f_1, f_2, \ldots, f_n\}$ and a set of points S we have

$$F(S) = \bigcup_{f \in F} f(S). \tag{12.2–1}$$

12.2.1 Sets of Affine Maps

The data type for an IFS is straightforward: an expression that contains a list of affine maps as element. We use the representation `ifs[{`*maps*`}]`. The constructor `IFS[{`*maps*`}]` creates an `ifs` object. To apply a list of maps to an object, we can use `Through[{`*maps*`}, `*obj*`]`. If the object is itself a list, we use mapping to effectively treat an IFS as listable. Graphic objects need special treatment because the result of `Through[{`*maps*`}, -Graphics-]` is a list of graphics. We should combine their ingredients into a single graphic. Here is the code for the application of an IFS to objects:

```
ifs[ms_List, _][gr:Graphics[_, opts___]] :=
    Graphics[ First /@ Through[ms[gr]], opts ]
(i_ifs)[objs_List] := i /@ objs
ifs[ms_List, _][obj_] := Through[ ms[obj] ]
```

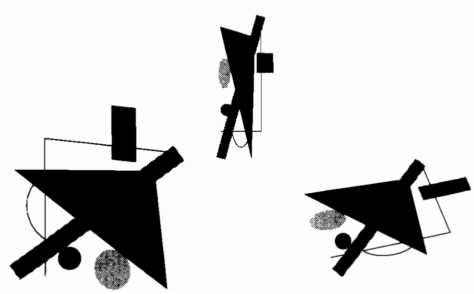
Part of IFS.m: Application of an IFS to objects

This sample IFS uses three of the maps from the example on page 314.

```
In[1]:= ifs0 = IFS[{
            scale[0.3, 0.8],
            AffineMap[0.2, 0.4, 0.9, 0.5, 2, -2],
            AffineMap[{x, y}, {0.9 y - 2, x - 0.1 y - 2}]
          }]

Out[1]= -ifs-
```

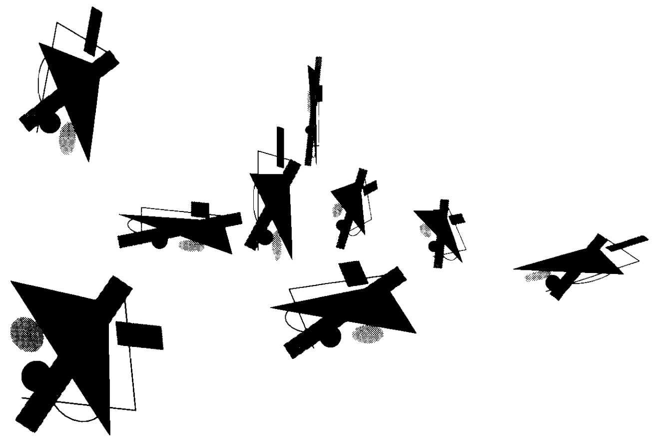
We apply it to the sample graphic from Section 12.1.

```
In[2]:= Show[ ifs0[gr1] ];
```



The main idea is to *iterate* the application. More and more images appear.

```
In[3]:= Show[ ifs0[%] ];
```



For the chaos game (to be described in Section 12.2.3) we must allow an IFS to take options. The idea is the same as is used in the data type `Graphics[]`. The options are stored as a second element, after the graphic primitives, in the form `Graphics[{`*primitives*`}, {`*options*`}]`. There is one difference in our treatment of an IFS: If no options are given when an IFS is defined, we insert the current default values of the options into the IFS object. In this way, an IFS will always have exactly two elements. This property simplifies some of our code. The code to insert the missing options is shown in Listing 12.2–1.

■ 12.2.2 Invariant Sets

An affine map is a contraction, if the absolute values of the linear contraction factors in all directions are less than 1. (This condition is equivalent to the requirements that all eigenvalues of the Jacobian matrix must be smaller than 1.) If all maps in an IFS F are contractions, there exists a unique set S of points in the plane that is invariant under F, that is,

$$F(S) = S\,. \tag{12.2–2}$$

Furthermore, this invariant set can be found by iteration. Start with any bounded set S_0 in the plane. The sequence of point sets S_0, S_1, S_2, ..., where $S_{i+1} = F(S_i)$, converges toward the invariant set. To make the notion of convergence precise, we need to define the metric we use to measure the distance between point sets. This theory is explained clearly in [34]; instead of repeating it here, let us give another example of contraction maps and fixed points: functions of real numbers.

Here is a simple real-valued function.

```
In[4]:= f[x_] := 1 + x/2
```

```
BeginPackage["ProgrammingInMathematica`IFS`",
             "ProgrammingInMathematica`AffineMaps`"]

IFS::usage = "IFS[{maps..}, {options..}] generates an iterated
    function system (IFS)."

ifs::usage = "-ifs- represents an iterated function system (IFS)."

Probabilities::usage = "Probabilities -> {pr..} is an option of IFS
    that gives the probabilities of the maps for the chaos game."

Options[ IFS ] = {
    Probabilities -> Automatic
};

Begin["`Private`"]

Format[ _ifs ] := "-ifs-"

(* Freeze missing options *)

optnames = First /@ Options[IFS]

IFS[ ms:{_map...}, opts___?OptionQ ] :=
    Module[{optvals},
        optvals = optnames /. Flatten[{opts}] /. Options[IFS];
        ifs[ ms, Thread[optnames -> optvals] ]
    ]

(* apply *)

ifs[ms_List, _][gr:Graphics[_, opts___]] :=
    Graphics[ First /@ Through[ms[gr]], opts ]

(i_ifs)[objs_List] := i /@ objs

ifs[ms_List, _][obj_] := Through[ ms[obj] ]

End[ ]

Protect[ IFS, ifs, Probabilities ]

EndPackage[ ]
```

Listing 12.2–1: IFS.m: Iterated function systems

The derivative of f is smaller than 1 everywhere.

```
In[5]:= D[ f[x], x ]

        1
Out[5]= -
        2
```

As a consequence, f has a unique fixed point.

```
In[6]:= Solve[ f[x] == x ]

Out[6]= {{x -> 2}}
```

The iteration $x_{i+1} = f(x_i)$ converges toward this fixed point for any choice of initial value x_0. Here, we chose $x_0 = 5.5$ and performed 15 iterations.

```
In[7]:= NestList[f, 5.5, 15]

Out[7]= {5.5, 3.75, 2.875, 2.4375, 2.21875, 2.10937,
  2.05469, 2.02734, 2.01367, 2.00684, 2.00342, 2.00171,
  2.00085, 2.00043, 2.00021, 2.00011}
```

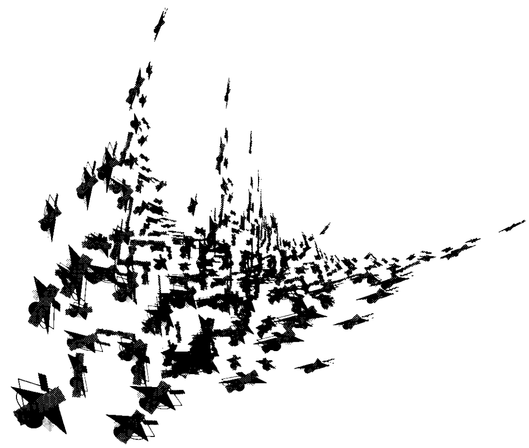
`FixedPoint[]` performs as many iterations as are necessary to find the fixed point within machine precision.

```
In[8]:= FixedPoint[ f, 5.5 ]

Out[8]= 2.
```

We can perform a similar iteration with an IFS. The initial set S_0 can be any set of points in the plane. We can use a graphic object to specify the set. It consists of all points that are part of one of the graphic primitives in the object. Let us use our sample graphic `gr1`.

Iteration of the IFS can be realized easily with `Nest[]`. Here we iterate the application of `ifs0` five times.

```
In[9]:= Show[ Nest[ifs0, gr1, 5] ];
```



It is not easy to describe the invariant set of our random collection of affine maps. Here is a more regular set of maps whose invariant set is easy to find exactly.

Here is a simple IFS with four maps. The maps are contractions by a factor $1/2$ composed with a translation.

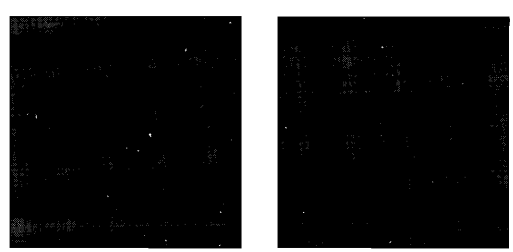
```
In[10]:= ex1 = IFS[{
           AffineMap[{x, y}, {0.5x, 0.5y}],
           AffineMap[{x, y}, {0.5x + 0.5, 0.5y}],
           AffineMap[{x, y}, {0.5x, 0.5y + 0.5}],
           AffineMap[{x, y}, {0.5x + 0.5, 0.5y + 0.5}]}];
```

Here is the unit square, rendered with a visible border.

```
In[11]:= square = Graphics[
          {{GrayLevel[0.6], Rectangle[{0,0}, {1,1}]},
           {Thickness[0.005],
            Line[{{0,0}, {0,1}, {1,1}, {1,0}, {0,0}}]}} ];
```

The left picture is the original square; on the right is the result of applying the IFS to the square. The unit square is an invariant set. The four smaller copies fit exactly into the original square.

```
In[12]:= Show[ GraphicsArray[{square, ex1[square]}] ];
```



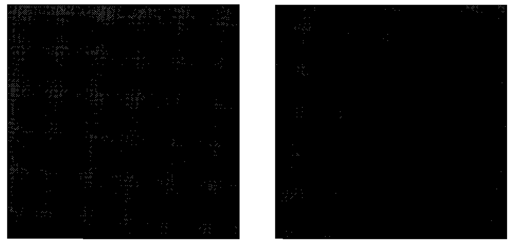
The condition that all linear contraction factors be smaller than 1 is necessary, as the next example shows.

Here is a simple IFS with two maps. The contraction factors in the vertical direction are equal to 1, not smaller than 1.

```
In[13]:= ex2 = IFS[{
             AffineMap[{x, y}, {0.5x, y}],
             AffineMap[{x, y}, {0.5x + 0.5, y}]} ];
```

The unit square is still an invariant set.

```
In[14]:= Show[ GraphicsArray[{square, ex2[square]}] ];
```



There are infinitely many invariant sets, however. Any rectangle with end points $\{0, 0\}$ and $\{1, r\}$ is invariant. Here, $r = 0.3$.

```
In[15]:= rect = Graphics[{Rectangle[{0,0}, {1,0.3}]}];\
         Show[ GraphicsArray[{rect, ex2[rect]}] ];
```

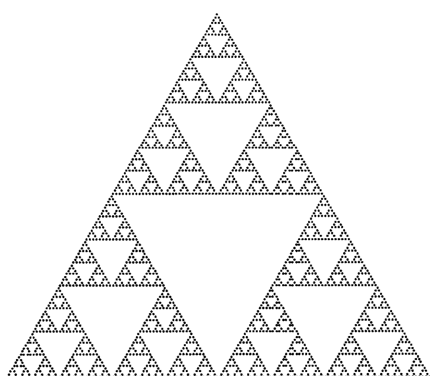
For many IFS the invariant set is a fractal. Here is the most famous example, the Sierpiński gasket.

The three maps in this IFS are scalings by $1/2$ composed with a translation. The three translation end points form a regular triangle.

```
In[16]:= sierpinski = IFS[{
             scale[0.5],
             Composition[scale[0.5], translation[{1, 0}]],
             Composition[scale[0.5],
                         translation[{1/2, Sqrt[3]/2}]]
         }];
```

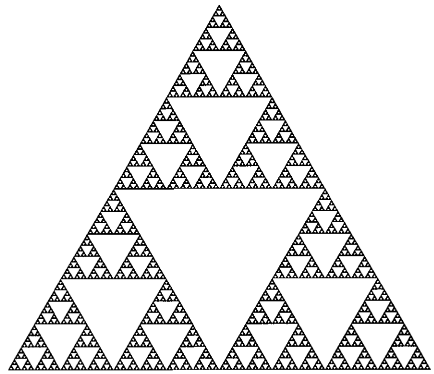
Here we start the iteration with a very simple graphic object: a single point. The IFS is iterated seven times.

```
In[17]:= Show[
         Nest[sierpinski,
              Graphics[{PointSize[0.6], Point[{0,0}]}],
              7] ];
```



Here, we use a triangle as initial object. The invariant set is independent of our choice of initial graphic. Finite approximations, such as these two, may still show visible differences, however.

```
In[18]:= Show[
          Nest[sierpinski,
               Graphics[{Polygon[{{0,0}, {1,0},
                                   {0.5,Sqrt[3]/2}}]}],
               7] ];
```



■ 12.2.3 The Chaos Game

There is a simpler method to illustrate the invariant set: the *chaos game*. Start with a single point and transform it by a randomly chosen map of the IFS. Then select another random map and apply it to the new point, and so on. Finally, draw the collection of all points obtained in this way.

Here is the code that generates the list of points, given a list of maps, a list of probabilities, and the number of iterations to perform:

```
makePoints[ maps_List, probs_, ntry_ ] :=
  Module[{random, n = Length[maps], colors = colors0, next},
    With[{cumul = FoldList[Plus, 0.0, probs]},
         random := With[{rand = Random[]},
                        Position[cumul, r_ /; r > rand, {1}, 1][[1,1]] - 1] ];
    NestList[ Unevaluated[maps[[random]]], Point[{0, 0}], ntry ]
  ]
```

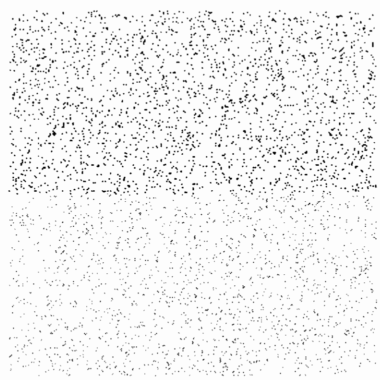
The variable `cumul` holds the cumulative probabilities (see Section 4.4.4.1). The result of evaluating `random` (the delayed assignment is important here) is an integer from 1 to `n`, the number of maps in the list. The list of points is produced with `NestList`. Note the use of `Unevaluated[]` to make sure that a random map is chosen at each step, rather than only once at the beginning. The function `makePoints` as it actually appears in ChaosGame.m is a bit more complicated than the version shown here, because we allow the option of coloring the points according to the map that generated them. The full code is shown in Listing 12.2–2.

The package ChaosGame.m is created automatically from the notebook ChaosGame.nb, which contains all commands that go into the package in initialization cells. Every time the notebook is saved, the package is updated automatically. This technique for developing a package in a notebook was explained in Section 11.1.1.

The command `ChaosGame[-ifs-, ntry, options]` plays the chaos game for the given IFS. Note that we throw away the first point, because it may lie far away from the invariant set. The option `Probabilities` allows the value of this option stored inside the IFS to be overridden. The option `PlotStyle` can be used to set the size and style of the points, and `Coloring` gives the coloring function to use. Note that `Probabilities` is not an option of `ChaosGame` itself, but of `IFS`. Nevertheless, we handle this option to allow the value stored in the IFS to be overridden. We implement the same operation that is used by `Show[]`: `Show[graphic, options]` accepts options on behalf of the graphic object given.

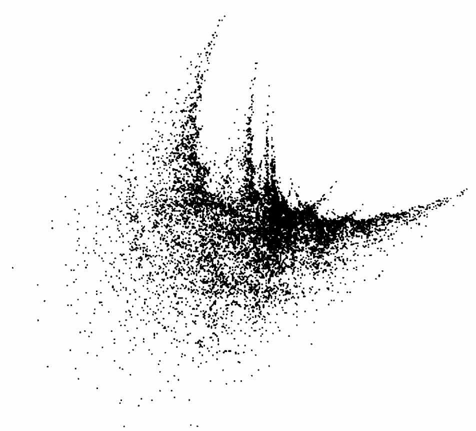
Because the invariant set is unique, the chaos game leads to the same limit set as does iteration (compare with Figure `Out[11]`). The value `Automatic` chooses different hues as colors. Colors can be used to illustrate how the maps in the IFS transform the invariant set.

```
In[19]:= ChaosGame[ ex1, 5000, Coloring -> Automatic ];
```



Here is a a rather unevenly distributed approximation of the invariant set of our introductory example, `ifs0`.

```
In[20]:= ChaosGame[ ifs0, 10000 ];
```



A more uniform distribution of the points can often be obtained by making the probabilities proportional to the contraction factors of the affine maps.

Here are the contraction factors of the three maps in `ifs0`.

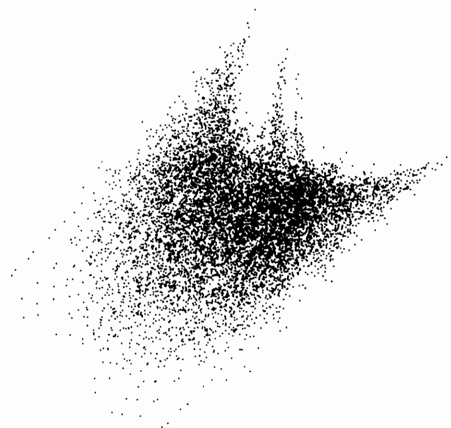
```
In[21]:= AverageContraction /@ ifs0[[1]]
Out[21]= {0.24, 0.44103, 0.9}
```

Here are the corresponding probabilities, normalized so that their sum equals 1.

```
In[22]:= ifsprobs = % / Plus @@ %
Out[22]= {0.1518, 0.278951, 0.569249}
```

The points are spread out more evenly with these probabilities.

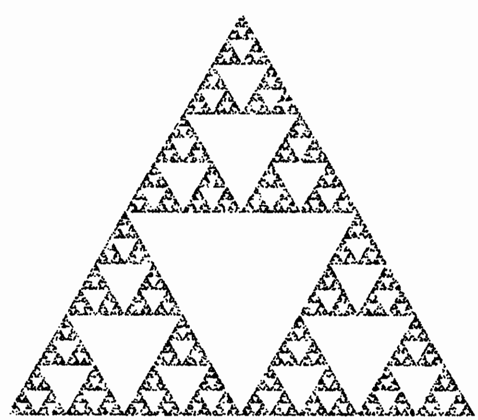
```
In[23]:= ChaosGame[ifs0, 10000, Probabilities -> ifsprobs];
```



There are more advanced methods than choosing the probabilities proportional to the contraction factors that lead to even more uniform pictures, see [19].

Here again is the Sierpiński gasket; compare with Figures `Out[16]` and `Out[17]`. On such regular fractals, a deterministic procedure gives usually better images than does a random chaos game.

```
In[24]:= ChaosGame[ sierpinski, 10000 ];
```



```
(*******************************************************************
This file was generated automatically by the Mathematica front end.
It contains Initialization cells from a Notebook file, which typically
will have the same name as this file except ending in ".nb" instead of
".m".

This file is intended to be loaded into the Mathematica kernel using
the package loading commands Get or Needs.  Doing so is equivalent to
using the Evaluate Initialiation Cells menu command in the front end.

DO NOT EDIT THIS FILE.  This entire file is regenerated automatically
each time the parent Notebook file is saved in the Mathematica front end.
Any changes you make to this file will be overwritten.
*******************************************************************)

BeginPackage["ProgrammingInMathematica`ChaosGame`",
             "ProgrammingInMathematica`IFS`"]

ChaosGame::usage = "ChaosGame[-ifs-, n, opts..] iterates random maps applied
    to a point n times and plots the result."
Coloring::usage = "Coloring -> val is an option of ChaosGame. Possible values are
    None, Automatic, a list of color directives, or a function of two
    arguments so that val[i, n] is the color of the ith out of n objects."
PlotStyle::usage = PlotStyle::usage <> " PlotStyle is an option of ChaosGame
    that specifies the style of the points."

Options[ ChaosGame ] = {
    PlotStyle -> PointSize[0],
    Coloring -> None };

Begin["`Private`"]

Needs["Utilities`FilterOptions`"]

ChaosGame::probs =
    "Probabilities `1` are not a list of nonnegative numbers of length `2`."

ChaosGame[ ifs[maps_List, ifsopts_], ntry_Integer?Positive, opts___?OptionQ ] :=
  Module[{pts, probs, ps},
    probs = Probabilities /. {opts} /. ifsopts;
    If[ probs === Automatic, probs = Table[1.0, {Length[maps]}] ];
    If[ Length[probs] != Length[maps] || !TrueQ[Plus@@probs > 0],
        Message[ChaosGame::probs, probs, Length[maps]]; Return[$Failed] ];
    probs = probs / Plus @@ probs; (* normalize, just in case *)
    ps = PlotStyle /. {opts} /. Options[ChaosGame];
    colorFunction = Coloring /. {opts} /. Options[ChaosGame];
    pts = makePoints[maps, probs, ntry, colorFunction];
    pts = Rest[pts]; (* drop first point *)
    Show[ Graphics[Join[Flatten[{ps}], pts]], FilterOptions[Graphics, opts],
          AspectRatio -> Automatic, PlotRange -> All ]
  ]

makePoints[ maps_, probs_, ntry_, colors0_ ] :=
  Module[{random, n = Length[maps], colors = colors0, next},
    With[{cumul = FoldList[Plus, 0.0, probs]},
         random := With[{rand = Random[]},
                        Position[cumul, r_ /; r > rand, {1}, 1][[1,1]] - 1] ];
    If[ colors === None || colors === False,
          NestList[ Unevaluated[maps[[random]]], Point[{0, 0}], ntry ]
        , (* else insert color directives *)
```

```
        If[ colors === Automatic, colors = Function[{i, k}, Hue[i/k]] ];
        If[ !ListQ[colors], colors = Table[colors[i, n], {i, n}] ];
        While[ Length[colors] < n, colors = Join[colors, colors] ]; (* cyclic *)
        With[{colors = colors},
            next[{_, p_}] := With[{i = random}, {colors[[i]], maps[[i]][p]}] ];
        NestList[ next, {GrayLevel[0], Point[{0, 0}]}, ntry ]
    ]
  ]

End[ ]
Protect[ ChaosGame, Coloring ]
EndPackage[ ]
```

Listing 12.2–2: ChaosGame.m: Randomly selected maps applied to a point

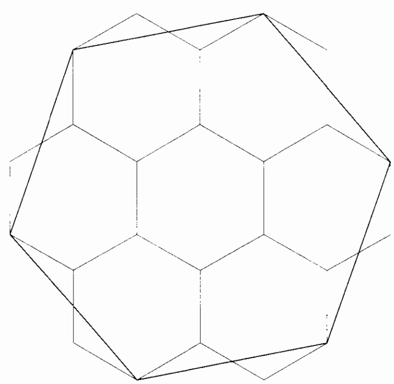
12.3 Examples of Invariant Sets

This section presents two interesting applications of IFS. First, we construct a hexagonal fractal tile, then we have a look at the use of IFS for the rendering of natural-looking pictures and image compression.

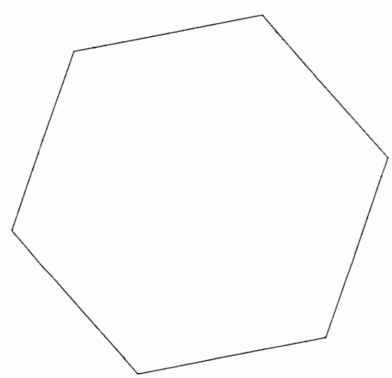
12.3.1 The Hexagonal Fractal Tile

Regular hexagons can be used to tile the plane, that is, nonoverlapping copies of hexagons can be arranged so as to cover the whole plane. The construction below shows how to find another tile with hexagonal symmetry that has the additional property that it can be cut into seven smaller copies of itself. Its boundary is a fractal curve.

This figure shows one step in the construction of the hexagonal tile. The large hexagon is subdivided into seven smaller ones that occupy the same area. The boundaries of the large hexagon and the collection of the seven small ones do not agree, however.



The process of subdivision can be iterated. Here is the third generation.



Let us determine the seven affine maps that map the large hexagon onto the seven smaller copies, respectively. The maps consist of a translation, a scaling, and a rotation. Once we have these seven maps, we can investigate the resulting IFS and its invariant set.

The seven midpoints of the small hexagons can be taken to be the origin and the sixth roots of unity. (The sixth roots of unity are equidistant points on the unit circle, just like the midpoints of the six outer hexagons.)

```
In[1]:= mids = Solve[x(x^6-1) == 0]

                                                          1/3
Out[1]= {{x -> -1}, {x -> 0}, {x -> 1}, {x -> -(-1)    },

               1/3               2/3                2/3
  {x -> (-1)    }, {x -> -(-1)    }, {x -> (-1)    }}
```

Here are the corresponding translations. The translation vectors consist of the real and imaginary parts of the seven complex points.

```
In[2]:= r7 = translation[{Re[#], Im[#]}]& /@ (x /. mids)

Out[2]= {-map-, -map-, -map-, -map-, -map-, -map-, -map-}
```

This is the required rotation angle. It can be deduced from the diagram on page 327 by elementary trigonometry.

```
In[3]:= hexarotation = ArcSin[Sqrt[3/7]/2];
```

The scaling factor is $1/\sqrt{7}$, because the linear scale is the square root of the area scale $1/7$ (seven small hexagons fit into the large one).

```
In[4]:= sf = 1/Sqrt[7];
```

This affine map properly scales and rotates the seven copies.

```
In[5]:= sr = Composition[ scale[sf],
                          rotation[hexarotation] ];
```

We compose it with the seven translations to get the seven maps for the IFS.

```
In[6]:= rmaps = Composition[#, sr]& /@ r7;
```

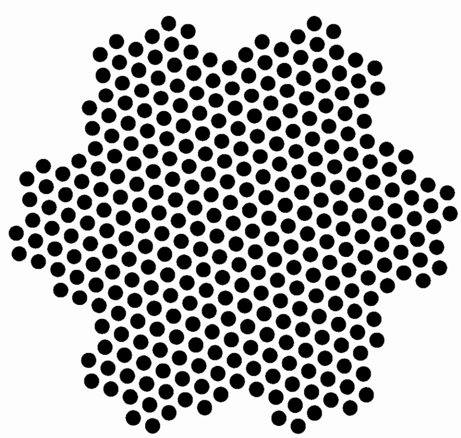
Here is the IFS for the fractal hexagonal tile.

```
In[7]:= hexatile = IFS[ N[rmaps] ];
```

Having found the seven maps, we can try to determine the invariant set of the corresponding IFS. There are two ways to do this: either start with an arbitrary graphic (here, a single point is a good choice) and iterate the IFS several times, or play the chaos game. The invariant set has the property that it can be cut into seven identical, smaller copies of itself. From this property it follows that it can also tile the plane.

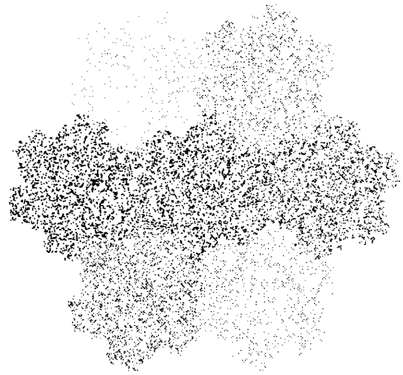
Here is an approximation, found by nesting the IFS three times. The initial graphic consists of just one (large) point. For a high-resolution version of this picture, see the chapter-opener graphic on page 215, where we nested the IFS six times.

```
In[8]:= Show[Nest[hexatile,
              Graphics[{PointSize[0.6], Point[{0,0}]}],
              3] ];
```



The chaos game gives a less satisfactory result for such a regular figure. However, with the chosen coloring, the seven smaller identical copies of the tile are clearly visible.

```
In[9]:= ChaosGame[ hexatile, 15000,
                    Coloring -> (GrayLevel[(#1-1)/#2]&) ];
```



■ 12.3.2 Images of Natural Objects

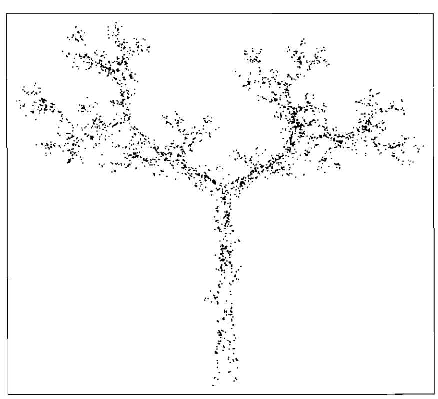
Many natural objects possess approximate self-similarity. Plants are an especially rich source of examples. Such self-similar objects can be modelled by an IFS with a small number of maps. The classical example is Barnsley's fern, reproduced in the chapter-opener picture (page 309). The image was produced with only four affine maps. The code is in the file BookPictures.m (Listing 10.3–1).

Here are five maps that describe an oak tree. The maps are given by their matrices.

```
In[1]:= oak = IFS[ AffineMap /@ {
          {{0.195,-0.488,0.4431},  {0.344,0.443,0.2452}},
          {{0.462,0.414,0.2511},  {-0.252,0.361,0.5692}},
          {{-0.058,-0.07,0.5976}, {0.453,-0.111,0.0969}},
          {{-0.035,0.07,0.4884}, {-0.469,-0.022,0.5069}},
          {{-0.637,0,   0.8562}, {0,      0.501,0.2513}} },
         Probabilities->{0.277,0.296,0.0416,0.0367,0.348}];
```

Here is the tree. The first few points are far away from the invariant set, but convergence to the set is fast.

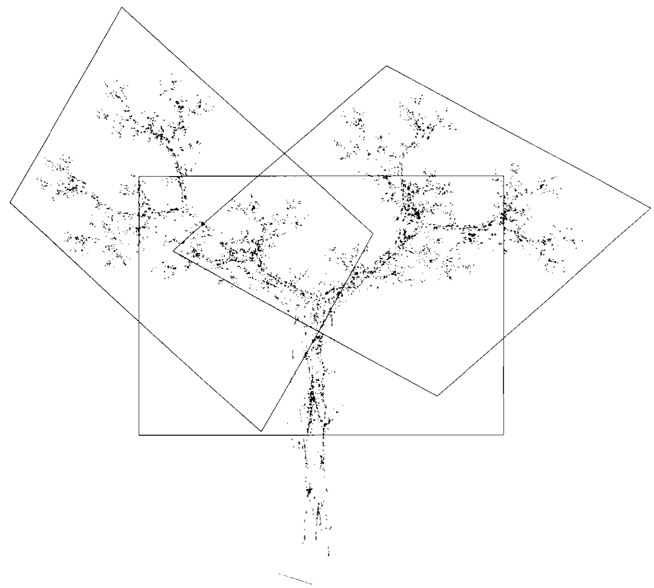
```
In[2]:= ChaosGame[oak, 2500, Frame->True, FrameTicks->None];
```



Here we show how the five images overlap. We take the previous picture (including the frame) and transform it under each of the five affine maps. Each part in one of the frames is a complete, affine copy of the whole.

`FullGraphics` takes a graphic object and expresses its appearance when rendered explicitly in terms of graphic primitives. In this way the frame, which is the result of the option setting `Frame->True`, is turned into an explicit closed line that can then be transformed by the IFS.

```
In[3]:= Show[ oak[FullGraphics[%]] ];
```



12.4 Documentation: Help Notebooks and Manuals

The help browser, available with some releases of Version 2.2, has been turned into a versatile tool for reading on-line documentation in all releases of *Mathematica*. The help browser can be configured with ordinary *Mathematica* syntax and it is extensible, so that you can add documentation for your own packages.

12.4.1 On-Line Documentation

The help documents are ordinary notebooks, of course. You can find the standard help notebooks in subdirectories of Documentation, arranged by language, for example, English. This directory, Documentation/English, contains subdirectories with the various categories of on-line help. Each of the subdirectories contains a file BrowserCategories.m, which the help browser uses to configure itself. Here, for example, is part of RefGuide/BrowserCategories.m:

```
BrowserCategory["Built-in Functions", None,
  {BrowserCategory["Numerical Computation", None,
    {BrowserCategory["Numerical Evaluation", None,
      {Item["N", "RefGuide.nb", MainEntry -> True]}],
     BrowserCategory["Equation Solving", None,
      {Item["Solve", "RefGuide.nb"], Item["NSolve", "RefGuide.nb"],
       Item["NDSolve", "RefGuide.nb", MainEntry -> True],
       Item["FindRoot", "RefGuide.nb"]}],
     BrowserCategory["Sums and Products", None,
      {Item["Sum", "RefGuide.nb"], Item["Product", "RefGuide.nb"],
        ⋮
      }]
    }],
   BrowserCategory["Calculus", None,
    {Item["D", "RefGuide.nb"], Item["Dt", "RefGuide.nb"],
     Item["Integrate", "RefGuide.nb", MainEntry -> True],
     Item["DSolve", "RefGuide.nb"],
     Item[Delimiter],
       ⋮
    }],
   ⋮
  }
]
```

Listing 12.4–1: RefGuide/BrowserCategories.m: Part of the browser description for the built-in functions

Such a browser category file has the structure

`BrowserCategory[`*name*`, `*directory*`, {`*entry*`, ...}].`

The *name* is the string that appears in the help browser listing. The *directory* is the subdirectory where the notebooks are to be found, or `None`, if the notebooks are in the same directory as is the browser category file. An *entry* can be one of the following

`Item[`*name*`,` *notebook*`,` *options*...`]`	an entry (hyperlink)
`BrowserCategory[ ... ]`	another category with subentries
`Item[Delimiter]`	a delimiter (horizontal line)
`HelpDirectoryListing[{`dir_1`, ...}]`	search directories for further browser category files

Help browser entries

If you select a *name* that belongs to a browser category, the entries in this subcategory appear to the right of the selected name. If you select a *name* that belongs to an item, the corresponding notebook is displayed in the viewing area of the help browser.

Because a notebook can be used to describe several items (the RefGuide.nb notebook that appears in the entries in Listing 12.4–1, for example, contains the entries for all built-in *Mathematica* functions), not all cells of the specified notebook are displayed by default, but only those that define *name* as one of their cell tags. A cell tag is a keyword that you can associate with a cell; the Find menu contains commands to view and set cell tags. If you want to display all cells, use the option `CopyTag->None`.

■ 12.4.2 Writing On-Line Documentation

The browser category file in Documentation/English/AddOns contains an entry

```
HelpDirectoryListing[AddOnHelpPath].
```

`AddOnHelpPath` is a frontend symbol whose value is a list of directories (you can examine it with the option inspector under Global Options ▷ File Locations). One of these directories is `FrontEnd`FileName[{$TopDirectory, "AddOns", "Applications"}]`.

`FrontEnd`FileName[{`*component*`, ...}]` is a system-independent representation of a path, used by the frontend. The components are maintained unevaluated; compare with the kernel function `ToFileName[{`*component*`, ...}]`, which turns the specification into a string. The symbol `$TopDirectory` gives your *Mathematica* installation directory.

The result is that the help browser will examine all packages installed in AddOns/Applications for a Documentation/English subdirectory containing a browser category file. In this way any installed packages with properly designed on-line help will be listed in the help browser. If you installed the packages for this book in

AddOns/Applications/ProgrammingInMathematica (see page xiv), a new item Programming in Mathematica should appear in the browser listing under the top-level category Add-ons.

Listing 12.4–2 shows the browser category file in ProgrammingInMathematica/Documentation/English. It points to (rudimentary) documentation for the packages developed in this chapter. All the documentation is in the notebook IFS.nb. Figure 11.1–2 shows the help browser displaying a topic from this notebook.

You must have the *Mathematica* on-line documentation installed to use the help browser. Note that the default location of the ProgrammingInMathematica directory is AddOns/ExtraPackages, rather than AddOns/Applications. The directory AddOns/ExtraPackages is probably not on the help search path `AddOnHelpPath`; see page xiv.

```
BrowserCategory["Programming in Mathematica", None, {
    Item["Overview", "Overview.nb", CopyTag->None],
    BrowserCategory["Affine Maps", None, {
        Item["AffineMap", "IFS.nb"],
        Item["AverageContraction", "IFS.nb"],
        Item["$CirclePoints", "IFS.nb"],
        BrowserCategory["Map constructors", None, {
            Item["rotation", "IFS.nb"],
            Item["scale", "IFS.nb"],
            Item["translation", "IFS.nb"]
        }]
    }],
    BrowserCategory["Iterated Function Systems", None, {
        Item["IFS", "IFS.nb"],
        Item["Probabilities", "IFS.nb"],
        Item["ChaosGame", "IFS.nb"],
        Item["Coloring", "IFS.nb"]
    }]
}]
```

Listing 12.4–2: ProgrammingInMathematica/Documentation/English/BrowserCategories.m

Here is a step-by-step guide to designing a help system for your package.

1. Decide on the hierarchical layout of your information and draft a corresponding browser category file.

2. Decide how the information is to be divided into help notebooks. For a smaller package you can put all information into a single notebook.

3. Write the help notebooks (this is the hard part. . .).

4. Add the names of the browser entries as cell tags in the help notebooks so that the correct cells are displayed for each browser entry. The menu command Find ▷ Add/Remove Cell Tags lets you edit cell tags. If an entry should display a whole notebook, use `CopyTag->None` in the corresponding browser category entry.

5. Install your package in a directory in AddOns/Applications and put the documentation (including the browser category file) into a subdirectory Documentation/English.
6. Invoke the menu item Help ▷ Rebuild Help Index so that the help browser becomes aware of the new information.
7. Test it out and rebuild the help index after making any changes to either the browser category file or one of the help notebooks.

Please note that official documentation from Wolfram Research on the help browser is not yet available at the time of this writing. Consult the *Programming in Mathematica Web Site* (see page xvi) for any new information as it becomes available.

■ 12.4.3 Package Installation

The directory AddOns/Applications is the standard place for third-party *Mathematica* applications. Each such application is put into a subdirectory AddOns/Applications/*myapplication*. In there, you can put the following items:

Packages: These are your *Mathematica* programs, residing in files with a .m extension. This book is mostly about what goes into these packages. The context name in the packages should be of the form `BeginPackage["`*myapplication*`` ` ``*file*`` ` ``"]` as explained in Section 2.6.

Auxiliary packages: Such packages can go into a subdirectory Common, as explained in Section 2.6.2.

Autoloading files: A file init.m or Kernel/init.m will be read when you give the command `Needs["`*myapplication*`` ` ``]`. As explained in Section 2.5.4, it can contain `DeclarePackage[]` commands to autoload all packages when the functions in them are used for the first time.

Notebooks: Notebooks with examples can go here, too, if they are not part of the on-line documentation. You can also decide to develop your packages in notebook form, as explained in Section 11.1.1. In this case, the .m packages will be generated automatically from the notebooks.

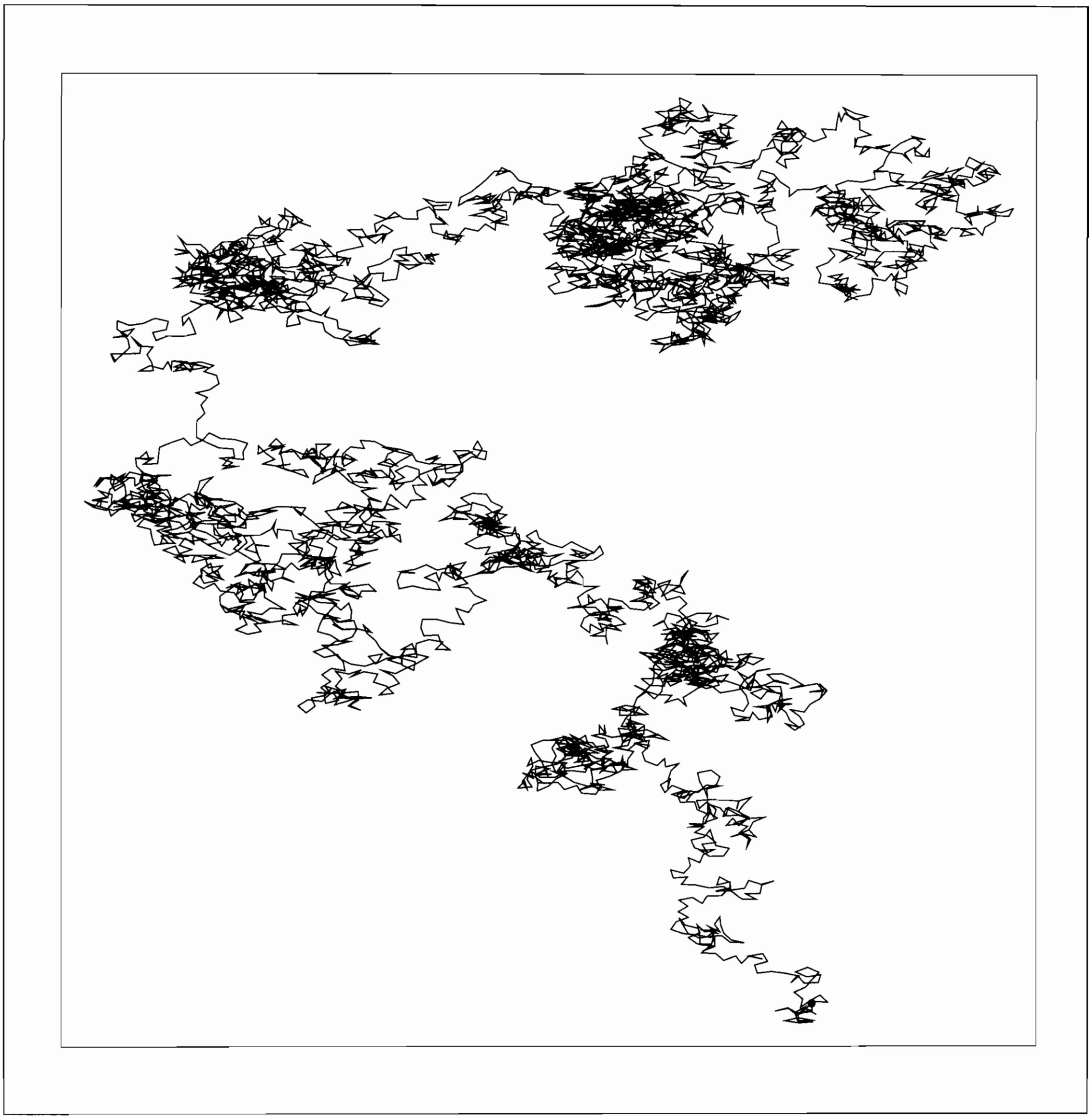
On-line documentation: The subdirectory Documentation/English contains the on-line documentation, as explained in Section 12.4.2.

Miscellanea: You can provide additional files, such as copyright notices, READMEs, and so on.

When you distribute a package you should instruct your users to install the files in the proper place, or provide an installation tool.

Appendix A

Exercises



This book should teach you enough about *Mathematica* that you can use the program to help you solve problems in your own area of research or teaching. Although the examples chosen always belong to a specific discipline of science or mathematics, you can use the same *methods* for your own purposes. The best exercise you can do to test your understanding of *Mathematica*'s programming language is to try to write a package that implements some of the algorithms you use in your own work.

Nevertheless, this appendix contains a few exercises related to the material covered in this book followed by sketchy solutions.

About the illustration overleaf:
A random walk. We start at the origin and then randomly choose a direction to follow for a line of length 1. This picture consists of 5000 segments. The code for the function `RandomWalk[]` is in the file RandomWalk.m, described in Section 4.4.3.

■ A.1 Programming Exercises

Most of the exercises require you to modify the code given in the book or to expand its functionality. The references in parentheses point to the place in the book where the topic of the exercise is covered. The difficulty is rated with a number from 0 to 10 in square brackets with 0 being trivial and 10 being impossible.

■ **1.** [4] Modify `CartesianMap[]` and `PolarMap[]` in the package ComplexMap.m so that the two sets of lines (horizontal and vertical or radial and angular) are displayed in different colors or gray levels (Chapter 1).

■ **2.** [6] Write a definition for expanding powers using the binomial formula

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}$$

that does not use any auxiliary variables (page 225).

■ **3.** [7] For the purpose of phototypesetting the manuscript for this book, the default width of lines in all kinds of graphics had to be changed. Find a way to do this for `Plot[]`, `Plot3D[]`, `ListPlot[]`, `Graphics[]`, and `Graphics3D[]`. The modifications should be transparent; no special command should be necessary when producing a graphic (Chapters 3 and 6).

■ **4.** [9] Find the fastest way of computing the n^{th} Fibonacci number (page 83).

■ **5.** [6] Write a package implementing the properties of the Dirac delta function δ used in mathematical physics (Chapter 8).

■ **6.** [6] There is a difference between the built-in `ReadList[]` and the function `MyReadList[]` from Section 9.2.6. If opening the file fails, `ReadList[]` returns itself unevaluated and `MyReadList[]` returns the empty list `{}`. Write a version of `MyReadList[]` that behaves like `ReadList[]` in this respect (page 247).

■ **7.** [3] The function `Explode[]` in Section 6.5.4 is defined as

```
Explode[atom_] := Characters[ ToString[InputForm[atom]] ].
```

Can you explain why we used `ToString[InputForm[atom]]` instead of simply `ToString[atom]` (page 188)?

■ **8.** [7] Write a definition for the format type `TeXForm` for tensors without looking at the package ProgrammingInMathematica`Tensors` (page 242).

■ **9.** [3] Write a command to plot a function together with its first n derivatives in one picture. It should accept options for `Plot[]` (Section 1.4.2).

■ **10.** [4] Use `Distribute[]` to generate a list of all divisors of a positive integer, from the number's prime factorization, obtained with `FactorInteger[]` (Section 4.7.5).

■ **11.** [8] The version of `ShowTime` described in Section 8.1.2 is not completely transparent. If you end an input with the semicolon (`;`) to prevent the display of the evaluation result, the value of the corresponding `Out[`n`]` line is not set to the (suppressed) result, but to `Null`. Therefore, you cannot refer to this result with `%`n. Modify ShowTime.m to remedy this defect.

■ **12.** [4] In many cultures the inverse trigonometric functions such as `ArcSin[`x`]` are denoted in traditional form by arcsin(x) instead of $\sin^{-1}(x)$ as they are in the USA. The inverse hyperbolic functions are denoted by arsinh(x) instead of $\sinh^{-1}(x)$. (Note: it is "arsinh," not "arcsinh." *Mathematica*'s use of `ArcSinh` is questionable.)

Write a package that sets up definitions for the formatting and interpretation of inverse trigonometric and hyperbolic functions in traditional form that follow these conventions (Section 9.5).

■ A.2 Solutions

We give short program fragments and hints only. It should be easy to turn those into a complete package if this is desired.

■ **1.** In the auxiliary procedure `Picture[]`, maintain the horizontal and vertical lines separately and insert an appropriate graphic primitive (`RGBColor[]` or `GrayLevel[]`, for example) at the beginning of each of these two sets of lines, then combine them.

■ **2.** Write the expression inside the `Sum[]` as a pure function and apply it to the range of integers from 0 to n, then add them together.

```
(a_ + b_)^n_Integer?Positive :=
    Plus @@ (Binomial[n, #] a^# b^(n-#)& /@ Range[0, n])
```

■ **3.** For `Plot[]` and `ListPlot[]`, you can set the default of the option `PlotStyle` to `Thickness[`*val*`]`. For `Plot3D[]` the option is `MeshStyle`.

For `Graphics[]` (and `Graphics3D[]`), the following rule will prepend the graphic primitive `Thickness[`*val*`]` to all graphics that do not have it already.

```
Graphics[l_List, rest___]/; Length[l] > 0 && Head[l[[1]]] =!= Thickness :=
    Graphics[Prepend[l, linewidth], rest]
```

■ **4.** There are formulae giving the n^{th} Fibonacci number directly that can be evaluated to a numerical approximation sufficient for rounding to the correct integer. The fastest method known to the author uses the fact that the power $n-1$ of the matrix

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

has the n^{th} Fibonacci number in the top left corner. A power tree method for computing matrix powers is very fast.

```
fib1[n_] :=
    Module[{ result = {{1,0},{0,1}}, new = n-1, m = {{1,1},{1,0}} },
        While[ new > 1,
            If[ OddQ[new], result = result . m ];
            new = Floor[new/2];
            m = m . m
        ];
        m[[1,1]] result[[1,1]] + m[[2,1]] result[[1,2]]
    ]
```

This program takes advantage of the fact that we need only one element from the last matrix computed. Because the last matrix multiplication is the one where the elements are the largest, this saves over half of the time needed otherwise (using the built-in `MatrixPower[]`).

We can also take advantage of the fact that all matrices involved are symmetric. Thus, we need to compute only three of the four elements. One of the three elements can be computed from the other two by a simple addition or subtraction; therefore, we need to compute only two of the elements using three multiplications instead of eight as the previous code does.

```
fib2[n_] :=
    Module[{ r11=1, r12=0, r22=1, new = n-1, m11=1, m12=1, m22=0 },
        While[ new > 1,
            If[ OddQ[new],
                {r11, r22} = m12*r12 + {m11*r11, m22*r22};
                r12 = r11 - r22 ];
            new = Floor[new/2];
            {m11, m22} = m12^2 + {m11^2, m22^2};
            m12 = m11 - m22;

        ];
        m11*r11 + m12*r12
    ]
```

Further identities can be used to bring the number of multiplications down to 2 inside the loop and a single one at the end; see [26]. With this method the computation of `fib[10^6]` on a SPARCstation 20 takes about 38 seconds. The result is approximately

$$1.9532821287077577316320149475962563324 \cdot 10^{208987}.$$

This method is now built into *Mathematica*.

■ **5.** The most important properties are the integral formulae

$$\begin{aligned}\int_{-\infty}^{\infty} \delta(x)\mathrm{d}x &= 1\\ \int_{-\infty}^{\infty} f(x)\delta(x-a)\mathrm{d}x &= f(a)\end{aligned}$$

which can be given as a single rule, using defaults:

```
Integrate[e_. DiracDelta[x_ + a_.], {x_,-Infinity,Infinity}] :=
    e /. x -> -a
```

The rule $\delta(x) = 0, x \neq 0$ is also straightforward:

```
DiracDelta[x_] /; x != 0 := 0.
```

This rule does not apply to symbolic arguments, which is important here!

■ **6.** You cannot simply say

```
Return[MyReadList[fileName, thing]]
```

inside `MyReadList[]` as this would lead to an infinite loop (the rule still matches). Instead, you can open the file as part of a side condition to the rule and simply let that condition fail if the open did not succeed. The rule will then not match and because there are no other rules for `MyReadList[]`, it will be left alone. The side condition can be put inside `Module[]` so that it can share local variables with the body of the rule. This is admittedly rather obscure. This feature is mentioned briefly in Subsection 2.6.1 of the *Mathematica* book.

```
MyReadList[fileName_String, thing_:Expression] :=
    Module[{file, expr, list = {}}, (
            While[ True,
                expr = Read[file, thing];
                If[ expr === EndOfFile, Break[] ];
                AppendTo[list, expr]
            ];
            Close[file];
            list
        ) /; (
            file = OpenRead[fileName];
            file =!= $Failed
        )
    ]
```

■ **7.** For symbols and integers it would indeed not make any difference. For other expressions `ToString[`*expr*`]` would give the string corresponding to the usual *two-dimensional* output form. It would be impossible to read this back in with `Intern[]`. If *expr* is a string itself, `ToString[`*expr*`]` would not include the quotation marks in the output string and `Intern[]` would convert the result back to a symbol rather than a string.

■ **8.** Because TeX does not like multiple subscripts or superscripts, we have to insert something in between them. Usually one uses `{}`, which produces no text.

```
Format[ Tensor[t_][ind___], TeXForm ] :=
    Block[{indices},
        indices = {ind} /. {ui->Superscript, li->Subscript};
        indices = Transpose[{Table["{}", {Length[indices]}], indices}];
        SequenceForm[t, Sequence @@ Flatten[indices, 1]]
    ]
```

We can still use the primitives `SubScript` and `Superscript` because *Mathematica* knows how to generate TEX output for subscripts and superscripts. Our example

```
TeXForm[Tensor[Gamma][li[k], ui[i], ui[j]][x, y, z, t]]
```

now becomes `{\rm Gamma}{}_{k}{}^{i}{}^{j}(x,y,z,t)`. When run through TEX it produces $\mathrm{Gamma}_k{}^{ij}(x, y, z, t)$ and we notice that we should have defined a TEX form for `Gamma` with

```
Format[Gamma, TeXForm] = "\\Gamma",
```

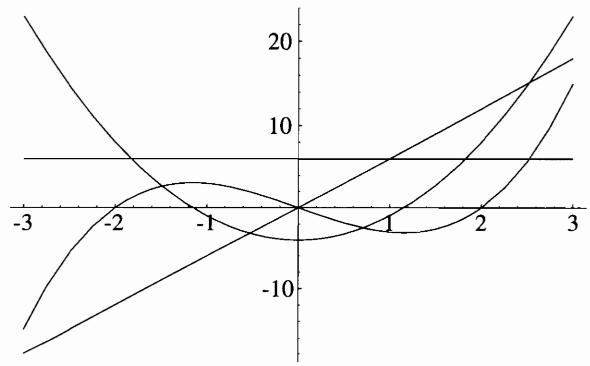
to get `\Gamma{}_{k}{}^{i}{}^{j}(x,y,z,t)` or finally $\Gamma_k{}^{ij}(x, y, z, t)$.

■ **9.** The syntax of the parameter list should be as close as possible to the one used in other plotting functions. We need an additional argument to specify the number of derivatives to plot. Using `NestList[]` instead of `Table[]` avoids the use of a loop variable. `Evaluate[]` is necessary, as we have seen in Section 5.3.3.

```
FunctionPlot[e_, range:{x_, __}, n_Integer?NonNegative, opts___?OptionQ] :=
    Plot[ Evaluate[NestList[D[#, x]&, e, n]], range, opts ]
```

Here is a third-degree polynomial together with its first three derivatives.

```
In[1]:= FunctionPlot[ x^3 - 4x, {x, -3, 3}, 3 ];
```



■ **10.** Let the prime factorization of n be

$$n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}\,.$$

The divisors of n are all numbers of the form $p_1^{e_1'} p_2^{e_2'} \cdots p_k^{e_k'}$, with $0 \le e_i' \le e_i$. All of these factors are distinct (this follows from the unique factorization theorem). The idea is now to generate the lists `{`p_i`^0,` p_i`^1, ...,` p_i`^`e_i`}` and then use `Distribute[]`. Here is an example, with $n = 3000$:

The result of `FactorInteger[]` is the list `{{`p_1`, `e_1`}, ..., {`p_k`, `e_k`}}`.

```
In[2]:= plist = FactorInteger[ 3000 ]
Out[2]= {{2, 3}, {3, 1}, {5, 3}}
```

The lists `{`p_i`^0, `p_i`^1, ..., `p_i`^`e_i`}` can be formed in this way, because `Power[]` is listable.

```
In[3]:= 5 ^ Range[0, 3]
Out[3]= {1, 5, 25, 125}
```

Here is a list of all the lists of prime powers.

```
In[4]:= Apply[ Function[{pi, ei},
                 pi^Range[0, ei]], plist, {1} ]
Out[4]= {{1, 2, 4, 8}, {1, 3}, {1, 5, 25, 125}}
```

Using `Distribute[]`, we pick one entry from each sublist and multiply them together. The outer operation is still `List`, because we want a list of the results. Sorting puts the divisors into a standard order.

```
In[5]:= Distribute[ %, List, List, List, Times ] // Sort
Out[5]= {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 25, 30,
  40, 50, 60, 75, 100, 120, 125, 150, 200, 250, 300, 375,
  500, 600, 750, 1000, 1500, 3000}
```

The comparison with the built-in divisor function confirms our result.

```
In[6]:= % == Divisors[3000]
Out[6]= True
```

■ **11.** The solution is surprisingly simple. The idea is this definition for inputs that end in a semicolon `;`:

```
ShowTime[ expr_; ] := (ShowTime[expr];)
```

To make it work for longer compound expressions (e_1`;` ...`;` e_n`;`), use this code instead:

```
ShowTime[ expr___; ] := (ShowTime[CompoundExpression[expr]];)
```

■ **12.** An expression of the form f`[`x`]` is typeset as *fp*(*xp*) in traditional form, where *fp* is the traditional name of f, and *xp* is the traditional form of x. The box structure of *fp*(*xp*) is `RowBox[{"`*fp*`", "(", `*xp*`, ")"}]`, where *xp* is the box structure for x. For `ArcSin`, for example, this rule defines the format:

```
MakeBoxes[ArcSin[x_], form:TraditionalForm] :=
    RowBox[{"arcsin", "(", MakeBoxes[x, form], ")"}];
```

The interpretation needs merely to turn `RowBox[{"arcsin", "(", x_, ")"}]` into `RowBox[{"ArcSin", "(", x, ")"}]`, which *Mathematica* already knows how to convert:

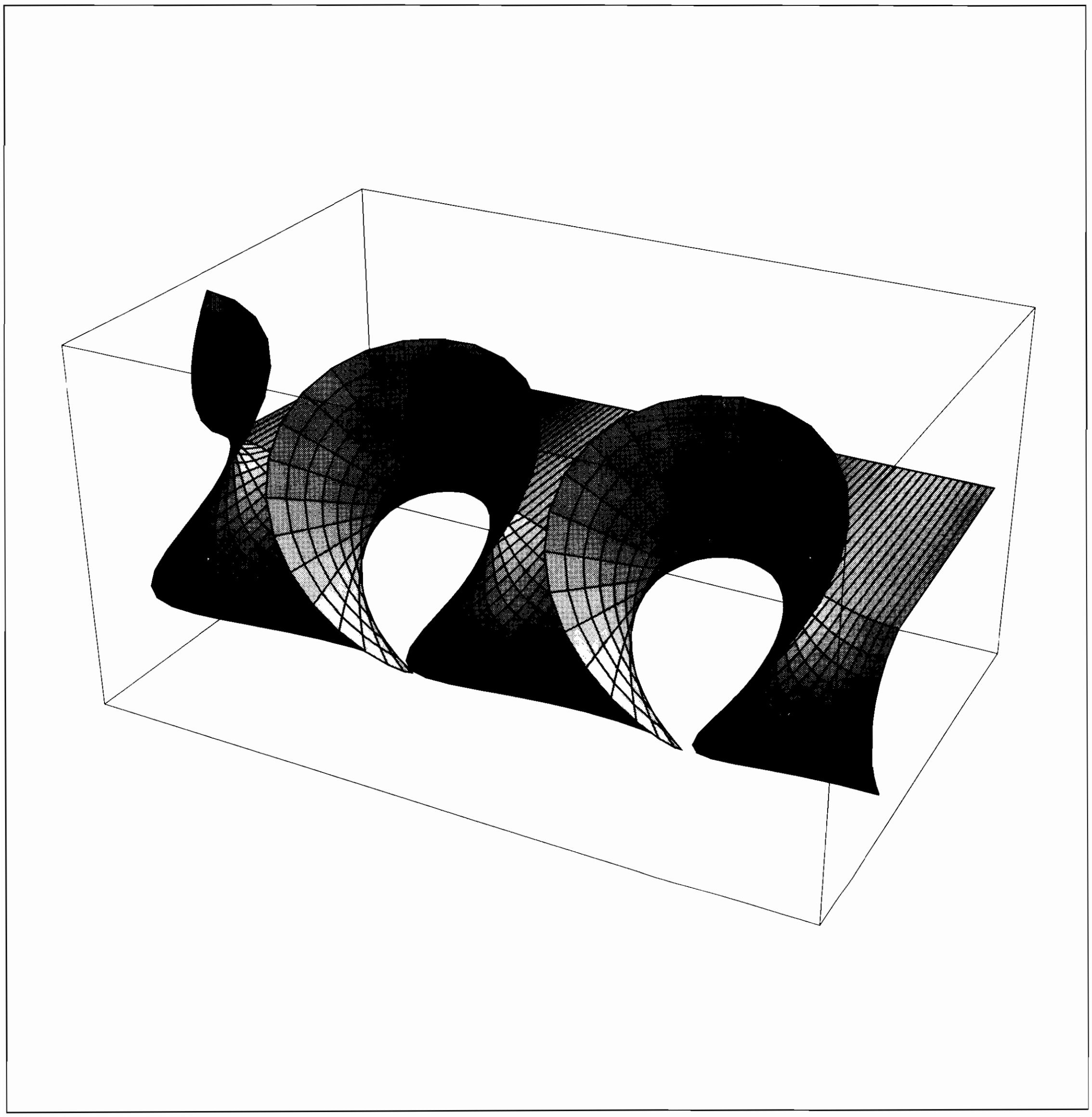
```
MakeExpression[RowBox[{"arcsin", "(", x_, ")"}], form:TraditionalForm] :=
    MakeExpression[RowBox[{"ArcSin", "(", x, ")"}], form]
```

Because we need to define these rules for many functions, it makes sense to define a procedure that takes the *Mathematica* symbol, such as `ArcSin`, and the traditional name, such as `"arcsin"`, as parameters and sets up these rules. For the trigonometric functions, the traditional-form name of f is simply `ToLowerCase[ToString[f]]`, and for the hyperbolic functions we use `StringReplace` to turn `"Arc"` into `"Ar"` to get the correct name. Here is the package TrigFormats.m that defines these formats:

```
BeginPackage["ProgrammingInMathematica`TrigFormats`"]

Begin["`Private`"]

arctrig = {ArcCos, ArcCot, ArcCsc, ArcSec, ArcSin, ArcTan};
artrigh = {ArcCosh, ArcCoth, ArcCsch, ArcSech, ArcSinh, ArcTanh};

defTraditional[arctrig_Symbol, name_String] :=
    With[{string = ToString[arctrig], form=TraditionalForm},
        MakeBoxes[arctrig[x_], form] :=
            RowBox[{name, "(", MakeBoxes[x, form], ")"}];
        MakeExpression[RowBox[{name, "(", x_, ")"}], form] :=
            MakeExpression[RowBox[{string, "(", x, ")"}], form]
    ]

defTraditional[#, ToLowerCase[ToString[#]]]& /@ arctrig
defTraditional[#, ToLowerCase[StringReplace[ToString[#], "Arc"->"Ar"]]]& /@
    artrigh

End[]

EndPackage[]
```

Appendix B

Bibliography



The sections in this appendix list the references I consulted for this book and additional literature recommended for those readers who wish to know more about some of the topics covered here. I have also included references to the technical literature about these subjects. The full bibliographic references follow at the end.

About the illustration overleaf:
Another minimal surface (see also the picture for Chapter 2). It is rendered as a parametric surface with the command

```
ParametricPlot3D[
   {r^2 Cos[2phi]/2 - Log[r], -phi - r^2 Sin[2phi]/2, 2 r Cos[phi]},
   {r, 0.0004, 2}, {phi, -2Pi, 3Pi}, PlotPoints -> {12, 100},
   ViewPoint->{-2.1, -1.1, 1.2} ]
```

B.1 Background Information and Further Reading

This section points to additional literature about several topics connected with the material in this book and lists the sources for some of my examples.

B.1.1 Programming Styles

Functional programming (Chapter 4) was introduced by the programming language LISP around 1960 [31]. A particularly clean implementation is the dialect SCHEME, developed at MIT [1, 38]. A delightful introduction to LISP is "the little LISPer" [13]. Starting with the basics, and a lot of humor in between, it gets right at the heart of the programming style typical of LISP. *Pure functions* (see Section 5.2) are central to LISP, where they are called *lambda expressions*. The theoretical foundation of functional programming is provided by the λ-calculus. See, for example, Barendregt's book [3].

Object-oriented programming took its origin with the language SIMULA [6]. It has become an important software design method. The most popular object-oriented language is SMALLTALK-80, like *Mathematica* an interpreted language [16]. Two recommended books about object-oriented software design are the ones by Meyer [32] and Booch [7].

The important concept of *modularization*—with its two key ideas *information hiding* and *interface declaration*— has been realized in many procedural languages, for example ADA and MODULA-2. It has also been added to implementations of other languages that did not include this idea from the beginning. The theoretical background was described by D. L. Parnas in 1972 [33].

I published articles on all major programming styles in the way they present themselves in *Mathematica*. Expanded and updated versions of these articles are part of the *Mathematica Programmer* book series [26, 27]. The role of pattern matching in programming is explained in [28].

Readers interested in the history of the development of programming languages should turn to the compilation by E. Horowitz of important papers about the origin of all major languages [22].

B.1.2 Numerical Analysis

For *numerical analysis* there is a vast body of literature. Most appropriate for users of *Mathematica* is the book by Skeel and Keiper [37]. *Numerical Recipes in C* by Press et al. [35] is a comprehensive source of more traditional procedural numerical programs. An excellent, more mathematically oriented treatment can be found in Froeberg's *Numerical Mathematics: Theory and Computer Applications* [14]. We encountered numerical computation in Chapter 7 and Section 8.4. Two often-used handbooks of mathematical functions are the one by Abramowitz and Stegun [2] and by Gradshteyn and Ryzhik [17].

■ B.1.3 Teaching with *Mathematica*

Mathematica is used more and more as a *teaching tool.* A complete college-level *calculus course* has been developed by Davis, Porta, and Uhl at the University of Illinois [10]. The motivation behind this course is described in an excellent article in the *Mathematica* Journal [8]. They make a strong point for using advanced software to teach a traditional mathematical subject. Other courseware has been developed; see the electronic *Mathematica* bibliography for a list of titles (Section B.1.6). For researchers in all sciences, *Mathematica* is a tool for doing their computations. Teaching the use of *Mathematica* should therefore emphasize *applications.* Courses about using symbolic computation to solve scientific problems have been taught at many universities. They can easily be adapted to *Mathematica.* Using just one program to do all the work greatly reduces the overhead involved in such courses (learning to use the computers and the various programs with their annoying differences in the user interface and input syntax). For a description of such courses, see [12]. A collection of projects can be found in [25]. Much can be gained at the high-school level or earlier by presenting mathematics as an *experimental adventure.* Given powerful, easy-to-use tools, such as *Mathematica*, students of all ages can discover many interesting mathematical facts in a playful, motivating way. Such explorative mathematics is the topic of a book by Gray and Glynn [18]. The journal *Mathematica in Education* [29] is devoted to all aspects of the use of *Mathematica* for teaching.

■ B.1.4 Various References

The Sierpiński sponge (page 353) has also been attributed to Menger. A picture similar to the one on the title page for the Index is reproduced in *Chaos: Making a New Science* by James Gleick [15, p. 101]. For *minimal surfaces*, see the articles by D. Hoffman [21] or S. Dickson [11]. The *Collatz* or $3x + 1$ problem (mentioned in Section 9.3.2) is further described in a survey article by J. Lagarias [23]. For a discussion of the Swinnerton-Dyer polynomials (Section 4.7.5), see Berlekamp's article [5]. The Lorenz attractor (Section 7.4.3) was described in [24]. The oscillator described in Section 7.4.4 is named after B. Van der Pol, who described it first in 1926 [39]. The nine regular polyhedra are described in Coxeter's *Regular Polytopes* [9]. We reproduced some of them in Section 4.6. A good introduction to chaos and fractals (Chapter 12) is [34]. More on fractal curves in *Mathematica* can be found in [26]. The use of iterated function systems for image compression was pioneered by Barnsley [4]. The IFS parameters for the figures on page 309 and in Section 12.3.2 have been taken from [34] and [20]. The Sierpiński gasket was first described in [36].

■ B.1.5 Literature on *Mathematica*

A list of the rapidly growing number of books on *Mathematica* is maintained by Wolfram Research; it is best accessed through the World Wide Web at the address given in the

following section. The *Mathematica Journal* [30] contains both news and explanatory articles about aspects of *Mathematica* and scholarly papers about the use of *Mathematica* as a research tool in the sciences. Another journal, *Mathematica in Education and Research* [29], is focused more toward applications in teaching.

■ B.1.6 Electronic Resources

The World Wide Web home page of Wolfram Research is the starting point for all *Mathematica*-related information on the Internet. It is at `<http://www.wolfram.com>`. My own electronic resources relating to this book are described on page xvi. The home page of *Programming in Mathematica* is at `<http://www.wolfram.com/Maeder/ProgInMath>`. There, you can find pointers to further electronic resources.

All programs described in this book are part of the *Mathematica*, Version 3, distribution from Wolfram Research; see page xiv.

B.2 References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, 1970.

[3] H. P. Barendregt. *The Lambda Calculus*. Studies in Logic 103. North Holland, revised edition, 1984.

[4] Michael F. Barnsley. *Fractals Everywhere*. AP Professional, second edition, 1993.

[5] E. R. Berlekamp. Factoring polynomials of large finite fields. *Math. Comp.*, 24:713–735, 1970.

[6] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Studentlitteratur Sweden, 1979.

[7] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991.

[8] Don Brown, Horacio Porta, and Jerry Uhl. Calculus&Mathematica: Courseware for the nineties. *The Mathematica Journal*, 1(1), 1990.

[9] H. S. M. Coxeter. *Regular Polytopes*. Dover Publications, Inc., 1973.

[10] Bill Davis, Horacio Porta, and Jerry Uhl. *Calculus&Mathematica*. Addison-Wesley, 1994.

[11] Stewart Dickson. Minimal surfaces. *The Mathematica Journal*, 1(1), 1990.

[12] Erwin Engeler and Roman Maeder. Scientific computation: The integration of symbolic, numeric and graphic computation. In B. Buchberger, editor, *Proceedings of Eurocal '85 Vol. I*, volume 203 of *SLNCS*, New York, 1985. Springer-Verlag.

[13] Daniel P. Friedman and Matthias Felleisen. *The little LISPer*. The MIT Press, 1987.

[14] Carl-Erik Froeberg. *Numerical Mathematics: Theory and Computer Applications*. The Benjamin/Cummings, 1985.

[15] James Gleick. *Chaos: Making a New Science*. Penguin Books, 1987.

[16] Adele Goldberg. *Smalltalk-80*. Addison-Wesley, second edition, 1989.

[17] Izrail Solomonovich Gradshteyn and Iosif Moiseevich Ryzhik. *Table of Integrals, Series and Products*. Academic Press, 1965.

[18] Theodore Gray and Jerry Glynn. *Exploring Mathematics with Mathematica*. Addison-Wesley, 1991.

[19] J. M. Gutiérrez, A. Iglesias, M. A. Rodríguez, and V. J. Rodríguez. Fractal image generation using iterated function systems. In V. Keränen, editor, *Mathematics with a Vision: Proceedings of the First International Mathematica Symposium*, pages 175–182. Computational Mechanics Publications, 1995.

[20] Dietmar Herrmann. Darstellung von Pflanzen mittels IFS. *PM Praxis der Mathematik*, (5), October 1993.

[21] David Hoffman. The computer-aided discovery of new embedded minimal surfaces. *The Mathematical Intelligencer*, 9(3), 1987.

[22] E. Horowitz, editor. *Programming Languages: A Grand Tour*. Computer Science Press, second edition, 1985.

[23] Jeffrey C. Lagarias. The 3x+1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, Jan. 1985.

[24] E. N. Lorenz. Deterministic nonperiodic flow. *J. Atmos. Sci.*, 20:130–141, 1963.

[25] Roman E. Maeder. A collection of projects for the mathematical laboratory. *SIGSAM Bulletin*, 21(3), August 1987.

[26] Roman E. Maeder. *The Mathematica Programmer*. AP Professional, 1994.

[27] Roman E. Maeder. *The Mathematica Programmer II*. Academic Press, 1996.

[28] Roman E. Maeder. Term rewriting and programming paradigms. In Troels Petersen, editor, *Elements of Mathematica Programming*, pages ??–??, TELOS/Springer-Verlag, 1996.

[29] *Mathematica in Education and Research*. TELOS/Springer-Verlag, 1996.

[30] *The Mathematica Journal*. Miller Freeman, Inc.

[31] John McCarthy. Recursive functions of symbolic expressions and their computation by machine I. *J. ACM*, 3:184–195, 1960.

[32] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[33] D. L. Parnas. A technique for software module specification with examples. *Commun. Ass. Comput. Mach.*, 15(5), May 1972.

[34] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals: New Frontiers of Science*. Springer-Verlag, 1992.

[35] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988.

[36] W. Sierpiński. Sur une courbe dont tout point est un point de ramification. *Compt. Rendus Acad. Sci. Paris*, 160:302–305, 1915.

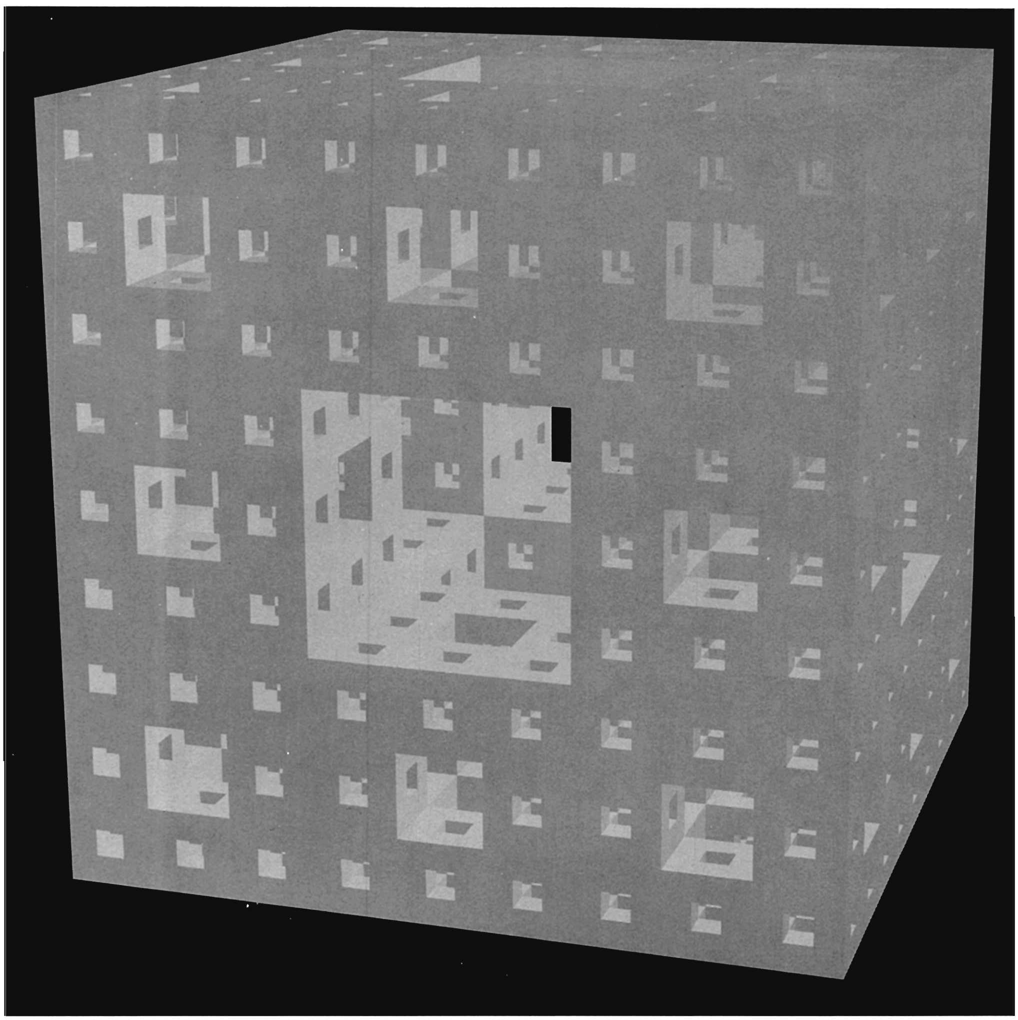
[37] R. Skeel and Jerry Keiper. *Elementary Numerical Computing with Mathematica*. McGraw-Hill, 1993.

[38] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. McGraw-Hill, 1989.

[39] B. Van der Pol. On relaxation oscillations. *Phil. Mag.*, 2:978–992, 1926.

[40] Stephen Wolfram. *The Mathematica Book*. Wolfram Media/Cambridge University Press, 3rd edition, 1996.

Index



The first index lists all programs mentioned in this book; it is followed by the usual index of topics and names.

About the illustration overleaf:
The third iteration of a *Sierpiński sponge*, a three-dimensional fractal figure. It is obtained by dividing a cube into 27 smaller cubelets and removing the six cubelets in the center of the faces and the one in the middle. Each of the remaining cubelets is then subdivided in the same way.

■ Programs

This table lists the page numbers of major program listings. Intermediate versions of packages are not shown. All programs are part of the *Mathematica*, Version 3, distribution from Wolfram Research; see page xiv.

■ Subjects and Names

Mathematica commands are listed in this index only if they are treated in some detail. On the other hand, you will find here all the commands developed in our example packages. These commands are not built in and are, therefore, not listed in the *Mathematica* book. The main entry for a topic is set in boldface. For typographical conventions, see page xv.

PROGRAMMING IN *MATHEMATICA*®

ROMAN MAEDER

Third Edition

This revised and expanded edition of the standard reference on programming in *Mathematica* addresses all the new features in the latest Version 3 of the software. The support for developing larger applications has been improved, and the book now discusses the software engineering issues related to writing and using larger programs in *Mathematica*. As before, Roman Maeder, one of the original authors of the *Mathematica* system, explains how to take advantage of its powerful built-in programming language.

New topics in this edition include:

- The programmable front-end;
- The language for typesetting mathematical expressions and the treatment of exact numerical quantities.
- A completely developed larger application, iterated function systems. This code allows readers to explore the fascinating world of chaos and fractals with *Mathematica*.

All programs discussed in this book are part of the Mathematica Version 3 distribution!

Current users of *Mathematica*, Version 2, and new users of Version 3 will benefit alike from this up-to-date reference to *Mathematica* programming.

Recommended by the present and past editors of *The Mathematica Journal*

"For anyone who wants to program in *Mathematica,* this book is the most thorough, systematic treatment available." — Troels Petersen

"This is an excellent book, and is getting better."
— Silvio Levy, founding editor, *The Mathematica Journal*

Roman Maeder was the third person to join the *Mathematica* development project, and was responsible for such parts of the system as polynomial factorization and language design. Maeder received his Ph.D. from the Swiss Federal Institute of Technology (ETH) in Zurich. Formerly a Professor of Computer Science at ETH, he is now an independent computing consultant.

9 780201 854497 53495

ISBN 0-201-85449-X

$34.95 US
$48.95 CANADA

ADDISON-WESLEY

Addison-Wesley is an imprint of Addison-Wesley Longman, Inc.

Text printed on recycled paper